Язык С

в XXI веке

Бен Клеменс



O'REILLY®

Бен Клеменс

Язык С в XXI веке

Ben Klemens

21st Century C



Бен Клеменс

Язык С в XXI веке



Москва, 2015

УДК 004.6
ББК 32.973.26
К48

Клеменс Бен
К48 Язык C в XXI веке / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2015. – 376 с.: ил.

ISBN 978-5-97060-101-3

Язык C – не просто фундамент всех современных языков программирования, он и сам – современный язык, идеальный для написания эффективных приложений передового уровня. Последние 20 лет C не стоял на месте. Сам язык и окружающая его экосистема подвергаются пересмотру. Эта книга начинается там, где другие заканчиваются. В ней рассказано, как изменилась функциональность, поддерживаемая любым компилятором, благодаря двум новым стандартам C, вышедшим со времен оригинального ANSI. Цель книги – рассмотреть то, чего нет в других учебниках по C: инструменты и окружение; библиотеки для работы со связанными списками и анализаторами XML; написание удобочитаемого кода с дружественным программным интерфейсом.

Издание предназначено для программистов, имеющих опыт работы на каком-либо языке и обладающими базовыми знаниями о C.

УДК 004.6
ББК 32.973.26

ISBN 978-1-491-90389-6 (анг.)
ISBN 978-5-97060-101-3 (рус.)

Содержание

Предисловие

Это ли истинный панк-рок,
Верный, как линия партии?

Wilco, «Too Far Apart»

Язык C и панк-рок

В языке C совсем немного **ключевых** слов, немало острых углов и безграничные возможности. Он позволяет сделать абсолютно все. Его изучение можно сравнить с гитарными аккордами C, G и D – основные движения освоить легко, а потом всю жизнь можно совершенствоваться. Отвергающие C боятся скрытой в нем мощи, считая ее небезопасной. По всем рейтингам C неизменно занимает первое место среди языков, продвижение которых не спонсируется никакими корпорациями или фондами[1].

Языку уже 40 лет, то есть он достиг среднего возраста. Ребята, создавшие его, сделали это вопреки желанию руководства – полная аналогия с истоками панк-рока, – но случилось это в 1970-х годах, и у языка было достаточно времени, чтобы стать популярным.

Что происходило, когда панк-рок стал популярным? Зародившись в 1970-х годах, панк, безусловно, вышел с обочины на большую дорогу. Тиражи альбомов таких групп, как The Clash, The Offspring, Green Day и The Strokes, исчисляются миллионами экземпляров, а в местном супермаркете мне доводилось слышать легкие инструментальные переложения песен в стиле отпочковашегося от панк-рока музыкального жанра «грандж». Бывший солист группы «Слиттер-Кинни» теперь ведет популярное комедийное шоу, в котором часто язвительно пародируются панк-рокеры[2]. В ответ на продолжающуюся эволюцию можно было бы занять жесткую позицию и сказать, что панк – это только то, что было в начале, а все остальное – легонький поп-панк для массовой аудитории. Блюстители традиций могут слушать свои пластинки 70-х годов, а когда бороздки износятся – скачать оцифрованное издание. А своих малолетних отпрысков одеть в «кенгурушки» – ностальгируя по группе «Рамонес».

Чужим этого не понять. Кто-то, слыша слово «панк», представляет себе канувшее в историю явление 1970-х годов – что-то связанное с парнями, которые делали нечто необычное. Традиционалисты, которые все еще любят и слушают диски Игги Попа, ловят свой кайф, но от того впечатление, что панк закостенел и уже неактуален, лишь усиливается.

---

[1] Это предисловие, вне всяких сомнений, многим обязано статье Криса Адамсона «Punk Rock Languages: A Polemic» по адресу http://pragprog.com/magazines/2011-03/punk-rock-languages.

[2] С такими стихами, как «Can't get to heaven with a three-chord song», быть может, Слитера-Кинни стоило отнести к постпанку? К сожалению, на панк нет стандарта ИСО, на который можно было бы ориентироваться, решая, кого куда отнести.

Однако вернемся в мир C. Тут тоже есть как традиционалисты, размахивающие знаменем со словами ANSI 89, так и люди, готовые использовать все, что реально работает, и даже не знающие, что код, который они пишут, в 1990-е годы нельзя было бы откомпилировать или запустить. Чужаки не замечают разницы. Они видят написанные в 1980-е книги, которые все еще лежат на прилавках, читают написанные в 1990-е онлайновые пособия, внимают упертым традиционалистам, которые настаивают, что и сегодня нужно писать, как тогда, и даже не понимают, что сам язык и его пользователи не застыли в развитии. Печально это – ведь они отказываются от поистине замечательных вещей.

Эта книга о том, как порвать с традицией и вернуть C новизну панк-рока. Я не собираюсь сравнивать свой код с оригинальной спецификацией, изложенной в книге Кернигана и Ричи 1978 года. В моем смартфоне 512 мегабайт памяти, так зачем же авторы учебников по C продолжают на десятках страниц наставлять, как сократить размер исполняемого файла на несколько килобайтов? Я пишу этот текст на дешевеньком нетбуке, способном выполнять 3 200 000 000 машинных команд в секунду, так какое мне дело до разрядности операндов команды: 8 или 16? Мы же в любом случае пишем на C, поэтому наш удобочитаемый, но неидеально оптимизированный код все равно будет работать на порядок быстрее, чем сравнимый код на любом другом распухшем от обилия функциональности языке.

Вопросы и ответы (или о параметрах этой книги)

В. Чем эта книга отличается от других книг по C?

О. Одни книги лучше написаны, другие даже занимательны, но у всех учебников по C есть одна общая особенность (а я прочитал их *множество*, в том числе [Deitel 2013], [Griffiths 2012], [Kernighan 1978], [Kernighan 1988], [Kochan 2004], [Oualline 1997], [Perry 1994], [Prata 2004] и [Ullman 2004]). По большей части, они были написаны до выхода стандарта C99, в котором упрощены многие аспекты использования языка. А бывает и так, что в очередное издание книги просто включено несколько замечаний о новшествах, но нет никакого серьезного переосмысления способов работы с языком. Во всех говорится, что, возможно, существуют библиотеки, которые могут пригодиться в собственном коде, но, как правило, ни слова об инструментах установки и экосистеме, благодаря которой эти библиотеки оказываются надежными и в разумной степени переносимыми. Материал, изложенный в этих учебниках, по-прежнему остается в силе и не утратил ценности, только вот современный код на C мало напоминает тот, который приводится в предлагаемых примерах.

Эта книга начинается там, где другие заканчиваются. Сам язык и окружающая его экосистема подвергаются пересмотру. Основная сюжетная линия – как пользоваться библиотеками для работы со связанными списками и анализаторами XML, а не разрабатывать собственные с нуля. Это книга о том, как писать удобочитаемый код с дружественным пользователю программным интерфейсом.

В. На кого рассчитана эта книга? На экспертов по кодированию?

О. Предполагается, что у вас есть опыт кодирования на каком-нибудь языке, к примеру на Java или скриптовом языке типа Perl. Я не собираюсь объяснять, почему программа не должна быть одной длинной процедурой, не разбитой на функции.

В тексте книги предполагается, что читатель обладает базовыми знаниями о C, приобретенными в процессе написания кода на этом языке. Для тех, кто подзабыл детали или вообще начинает с азов, в приложении A приводится краткий справочник по основам C, рассчитанный на владеющих такими скриптовыми языками, как Python или Ruby.

Наверное, стоит упомянуть, что я написал также учебник по статистическим и научным расчетам «Modeling with Data» [Klemens 2008]. Помимо многочисленных деталей, относящихся к численным методам и использованию статистических моделей для описания данных, там имеется более развернутое и независимое руководство по C, в котором, надеюсь, мне удалось преодолеть многие недостатки прежних руководств.

В. Я прикладной программист и не собираюсь копаться в ядре. Зачем мне писать на C, а не на скриптовом языке Python, на котором программировать куда быстрее?

О. Если вы прикладной программист, то эта книга как раз для вас. Сколько раз я слышал утверждение, будто C – язык системного программирования; оно страшно далеко от панковского склада ума – да кто они такие, чтобы указывать нам, что можно писать, а что – нет?

Высказывания типа «наш язык почти такой же быстрый, как C, но писать на нем проще» уже набили оскомину. Понятно же, что сам C такой же быстрый, как C, а задача этой книги – убедить вас в том, что писать на нем проще, чем это следует из книг десятилетней давности. Вызывать `malloc` и забираться в дебри управления памятью вам придется даже вполовину не так часто, как системному программисту 1990-х годов. У нас теперь есть простые средства для работы со строками, и даже базовый синтаксис изменился, чтобы сделать код понятнее.

Я всерьез начал писать на C, когда понадобилось ускорить программу моделирования, написанную на скриптовом языке R. Как и многие другие скриптовые языки, R имеет интерфейс к C, которым предлагается пользоваться всякий раз, как включающий язык оказывается слишком медленным. В конечном итоге я переписал на C так много функций, что от включающего языка вообще отказался.

В. То, что эта книга понравится прикладным программистам с опытом работы на скриптовых языках, конечно, прекрасно, но я-то занимаюсь кодом ядра. Я выучил C в пятом классе, у меня даже сны иногда успешно компилируются. Что тут для меня может быть нового?

О. Последние 20 лет C не стоял на месте. Ниже я расскажу о том, как изменилась функциональность, гарантированно поддерживаемая любым компилятором, – благодаря двум новым стандартам C, вышедшим со времен оригинального стандарта ANSI. Загляните в главу 10, может статься, кое-что в ней вас удивит.

В некоторых частях книги, например в главе 6, развенчивающей широко распространенные ошибочные представления об указателях, рассматриваются вещи, изменившиеся с 1980-х годов.

Прогресс затронул и окружение. Многие рассматриваемые мной инструменты, например make и отладчик, вам, наверное, знакомы, но есть другие, не столь хорошо известные. Комплект инструментов Autotools полностью изменил представление о распространении кода, а система управления версиями Git знаменует новый подход к коллективной разработке кода.

В. Не могу не отметить, что примерно в трети книги вообще нет кода на С.

О. Цель этой книги – рассмотреть то, чего нет в других учебниках С, а первым номером в этом списке стоят инструменты и окружение. Если вы не пользуетесь отладчиком (автономным или входящим в IDE), то заметно усложняете себе жизнь. Во многих учебниках отладчик вынесен куда-то на задворки, если вообще упоминается. Для совместной работы над кодом нужен другой комплект инструментов, включающий среди прочего Autotools и Git. Код существует не в вакууме, и я полагал, что окажу читателям дурную услугу, написав еще одну книгу, основанную на предпосылке, будто знание синтаксиса – это все, что необходимо для продуктивного использования языка.

В. Есть много средств для разработки программ на С. Какими критериями вы руководствовались при отборе?

О. Сообщество пользователей С в большей степени, чем многие другие, озабочено следованием стандартам интероперабельности. Существует масса расширений С, предлагаемых в среде GNU, есть интегрированные среды (IDE), которые работают только в Windows, а также расширения компилятора, доступные лишь в LLVM (Low Level Virtual Machine – низкоуровневая виртуальная машина). Быть может, именно поэтому авторы прежних учебников боялись затрагивать тему инструментальных средств. Но в наши дни существуют системы, которые работают на всем, что мы обычно считаем компьютером. Многие являются частью проекта GNU; LLVM со своим инструментарием быстро набирают популярность, но пока еще не являются преобладающими. Где бы вы ни работали – в Windows, в Linux, на экземпляре, только что полученном от поставщика облачных вычислений, – рассматриваемые здесь инструменты можно будет установить легко и быстро. Я упомяну о нескольких платформенно-зависимых инструментах, но всякий раз буду явно отмечать это.

Я не рассматриваю интегрированных сред разработки (IDE), потому что вряд ли найдутся такие, которые надежно работают на любой платформе (попробуйте поставить Eclipse и подключаемые к ней модули для С на экземпляр Amazon Elastic Compute Cloud), да к тому же выбор IDE в высшей степени субъективен. В состав типичной IDE входит система сборки проектов, которая обычно несовместима с аналогичной системой сборки из другой IDE. Поэтому файлы проектов IDE невозможно использовать для распространения проекта; исключением являются случаи, когда все заинтересованные лица обязаны работать с одной и той же IDE (учебные курсы, некоторые офисы, некоторые вычислительные платформы).

В. У меня есть Интернет. Чтобы посмотреть справку по команде или нюансы синтаксиса, хватит пары секунд, так зачем мне читать книгу?

О. Истинная правда: чтобы посмотреть таблицу приоритетов операторов в системе Linux или Mac, достаточно набрать команду `man operator`. Почему же тогда я привожу ее в книге?

У меня точно такой же Интернет, как у вас, и я немало времени провожу в нем, читая разные материалы. Поэтому я прекрасно знаю, о чем там не пишут, и это именно то, что я включил в книгу. Описывая новый инструмент, например gprof или GDB, я сообщаю то, что необходимо знать, чтобы сориентироваться и задать поисковой системе разумные вопросы, а также то, о чем умалчивают другие учебники (а это ой как много).

Стандарты: как много девушек хороших

Если явно не оговорено противное, весь код в книге соответствует стандартам ISO C99 и C11. Чтобы вы понимали, о чем идет речь, будет уместно дать краткий исторический обзор и перечислить основные стандарты языка C (опуская мелкие редакционные правки и исправления).

K & R (примерно 1978)

Деннис Ричи, Кен Томпсон и ряд сподвижников придумали язык для написания операционной системы Unix. Впоследствии Брайан Керниган и Деннис Ричи привели описание языка в первом издании своей книги. Это и был первый стандарт *де-факто* [Kernighan 1978].

ANSI C89

Компания Bell Labs передала курирование языка Американскому национальному институту стандартов (ANSI). В 1989-м был опубликован первый стандарт, содержавший ряд усовершенствований, по сравнению с K&R. Во второе издание книги K&R была включена полная спецификация языка, а это означало, что на рабочих столах десятков тысяч программистов появился экземпляр стандарта ANSI [Kernighan 1988]. В 1990 году стандарт ANSI был принят Международной организацией по стандартизации (ИСО) без существенных изменений, но ANSI 89 употребляется чаще (и встречается на футболках).

Прошло десять лет. Язык C стал общепринятым в том смысле, что базовый код практически всех ПК и всех интернет-серверов написан на C; пожалуй, трудно вообразить более «общепринятое» творение человеческого разума.

За это время от C откололся C++ и добился значительного успеха (хотя и не такого значительного, как C). Появление C++ стало лучшим, что когда-либо происходило с C. В то время как другие языки обзаводились дополнительными синтаксическими конструкциями, чтобы следовать в русле объектной ориентированности, и всякими другими фенечками, приходившими на ум их авторам, C строго придерживался стандарта. Те, кто хотел стабильности и переносимости, писали

на С. Те же, кому были нужны все новые и новые возможности, чтобы купаться в них, как в ванне с шампанским, получили в свое распоряжение C++. И все были счастливы.

ISO C99

Спустя десять лет С подвергся существенному пересмотру. Были добавлены средства для численных и научных расчетов, стандартный тип комплексных чисел и некоторые подобия обобщенных функций, адаптирующихся к типу аргументов. Включены некоторые удобные средства C++, включая однострочные комментарии (впервые появившиеся в предшественнике С – языке BCPL) и возможность объявлять переменные в заголовке цикла `for`. Упрощена работа со структурами – благодаря новым правилам объявления и инициализации и некоторым нотационным усовершенствованиям. Признано, что безопасность – не последнее дело и что не все в мире говорят по-английски, и это тоже оказало влияние на модернизацию языка.

Размышляя о том, сколько нового появилось в стандарте C89, и учитывая, что нет в мире компьютера, где бы не работал код на С, трудно представить, что ИСО мог придумать нечто такое, что не подверглось бы ожесточенной критике, – его ругали даже за отказ вносить те или иные изменения. И нельзя не признать, что стандарт оказался противоречивым. Существует два общепринятых способа представить комплексное число (в прямоугольных и в полярных координатах) – так почему ИСО отдал предпочтение только одному? Зачем нужен механизм макросов с переменным числом аргументов, если код прекрасно можно писать и без него? Иными словами, блюстители чистоты идеи обвиняли ИСО в том, что тот уступил давлению со стороны жаждущих новой функциональности. В настоящее время большинство компиляторов поддерживают C99 с некоторыми оговорками, например серьезные трудности вызывает тип `long double`. Однако есть одно заметное исключение из этого широкого консенсуса: корпорация Майкрософт отказывается включать поддержку C99 в свой компилятор Visual Studio C++. В разделе «Компиляция кода на С в Windows» ниже мы увидим некоторые из многочисленных способов откомпилировать код в Windows, так что отказ от Visual Studio – не более чем неудобство, а желание крупного рыночного игрока запретить нам использование стандартов ANSI и ISO только укрепляет дух панк-рока, свойственный С.

C11

Сознавая справедливость обвинений в уступке давлению, ИСО внес серьезные изменения в третью редакцию стандарта. Стало возможно писать обобщенные функции, дальнейшее развитие получили также идеи безопасности и интернационализации.

Стандарт C11 вышел в декабре 2011 года, но разработчики компиляторов на удивление быстро поддержали его. Сейчас большинство основных компиляторов заявляет о почти полном соответствии. Однако стандарт опреде-

ляет поведение не только компилятора, но и стандартной библиотеки, а вот поддержка библиотеки и, в частности, многопоточности и атомарных операций в одних системах реализована, а в других отстает.

Стандарт POSIX

Выше было описано положение дел с самим языком C, однако он всегда развивался вместе с операционной системой Unix, и в книге вы увидите, что эта взаимосвязь существенна для повседневной работы. Если какая-то задача легко решается с помощью командной строки Unix, то, скорее всего, она легко решается и на C; инструментальные средства Unix часто создаются, чтобы упростить кодирование на C.

Unix
C и Unix были разработаны в компании Bell Labs в начале 1970-х годов. На протяжении большей части XX столетия Bell постоянно преследовалась за монополистическую практику, и в одном из соглашений с правительством США компания пообещала не распространять свои коммерческие интересы на разработку программного обеспечения. Таким образом, система Unix была бесплатно отдана исследователям, которые получили право разбирать ее на части и собирать заново. Само слово Unix является торговым наименованием, которое первоначально принадлежало Bell Labs, а затем разошлось, как бейсбольная карточка, среди многочисленных компаний.

Варианты Unix плодились один за другим по мере изучения, переделки и улучшения кода независимыми энтузиастами. Достаточно было одной крохотной несовместимости, чтобы сделать программу или скрипт непереносимыми, поэтому очень скоро была осознана необходимость какой-то стандартизации.

POSIX
Этот стандарт, впервые выпущенный Институтом инженеров электротехники и электроники (IEEE) в 1988 году, заложил общую основу для всех Unix-подобных операционных систем. В нем описывается, как должна работать оболочка, чего ожидать от команд типа `ls` и `grep`, а также ряд библиотек, на которые имеют право рассчитывать программисты, пишущие на C. Например, именно здесь детально описан механизм конвейеров, с помощью которых сцепляются команды в оболочке; это означает, что библиотечная функция `popen` (открыть конвейер) описана в стандарте POSIX, а не ISO C. Стандарт POSIX много раз пересматривался; на момент написания этой книги последней является версия POSIX:2008, и именно ее я имею в виду, говоря, что нечто совместимо с POSIX. В системе, соответствующей стандарту POSIX, должен присутствовать компилятор C, доступный по имени `c99`.
В этой книге стандарт POSIX используется, и я скажу об этом, когда дело до него дойдет. За исключением многочисленного семейства ОС от Майкрософт, практически все прочие операционные системы совместимы с POSIX: Linux, Mac OS X, iOS, webOS, Solaris, BSD, даже в серверных ОС Windows имеется

подсистема POSIX. А о том, как установить подсистему POSIX для систем, выбивающихся из общего ряда, написано в разделе «Компиляция кода на C в Windows» ниже.

Наконец, существуют еще две реализации POSIX, о которых стоит упомянуть ввиду их широкой распространенности и влиятельности.

BSD
После того как компания Bell Labs отдала Unix на растерзание публике, ученые из Калифорнийского университета в Беркли внесли существенные усовершенствования и в конечном итоге полностью переписали код Unix, в результате чего появился дистрибутив Berkeley Software Distribution. Всякий, кто пользуется компьютером производства компании Apple, работает с системой BSD, снабженной привлекательным графическим интерфейсом. В некоторых отношениях BSD выходит за рамки POSIX, и мы увидим функции, которые в стандарт POSIX не входят, но настолько полезны, что обойти их вниманием никак нельзя (и прежде всего такая палочка-выручалочка, как `asprintf`).

GNU
Этот акроним расшифровывается как «GNU is Not Unix», это еще один успешный пример независимой реализации и улучшения окружения Unix. В подавляющем большинстве дистрибутивов Linux используются инструментальные средства GNU. Почти наверняка на вашем POSIX-совместимом компьютере установлен набор компиляторов GNU Compiler Collection (gcc), даже в BSD он есть. Подчеркнем, что gcc – это стандарт де-факто, который в некоторых направлениях расширяет C и POSIX. Всюду, где эти расширения встречаются, я буду упоминать о них явно.

С юридической точки зрения, лицензия BSD чуть более либеральна, чем лицензия GNU. Поскольку некоторые организации глубоко озабочены политическими и деловыми аспектами лицензий, большинство инструментов предлагается в вариантах для GNU и для BSD. Например, как в gcc, так и в BSD имеется проект `clang`, включающий первоклассные компиляторы языка C. Разработчики из обоих лагерей пристально наблюдают друг за другом и перенимают достижения, поэтому можно ожидать, что существующие на сегодня различия в будущем сойдут на нет.

Юридические вопросы

В законодательстве США больше нет системы регистрации прав на интеллектуальную собственность; за немногими исключениями, все, что кем-то написано, автоматически защищено таким правом.

Разумеется, для распространения библиотеки необходимо копирование с одного жесткого диска на другой, и существует целый ряд общеупотребительных механизмов предоставления права копирования произведения, защищенного правом интеллектуальной собственности, с минимумом формальностей.

Лицензия *GNU Public License* разрешает копирование исходного кода и его исполняемого варианта без ограничений. Существует одно важное условие: если вы *распространяете* программу или библиотеку, в которой используется исходный код, защищенный

лицензией GPL, то обязаны распространять и исходный код своей программы. Следует понимать, что если вы используете программу только внутри организации и не распространяете ее, то это условие к вам не относится, и включать в дистрибутив исходный код необязательно. Исполнение программ, распространяемых на условиях GPL, например компиляция своего кода с помощью gcc, не влечет обязательств по распространению своего исходного кода, потому что результат работы программы (например, созданный компилятором исполняемый файл) не считается производным продуктом gcc. [Пример: библиотека GNU Scientific Library.]

Лицензия *Lesser GPL* во многом аналогична GPL, но при этом явно оговаривается, что если вы компонуете свою программу с разделяемой библиотекой, распространяемой на условиях LGPL, то ваш код не считается производным продуктом, и потому распространять исходный код необязательно. Иначе говоря, разрешено распространять без исходного кода программу, скомпонованную с LGPL-библиотекой. [Пример: библиотека GLib.]

Лицензия *BSD* требует сохранения авторских прав и оговорок об ограничении ответственности для распространяемого на условиях этой лицензии исходного кода, но не требует включения исходного кода в дистрибутив. [Пример: библиотека Libxml2, распространяемая по лицензии MIT, аналогичной BSD.]

Примите во внимание обычную оговорку: я не юрист, и это всего лишь краткое изложение довольно длинных юридических документов. Если вы не уверены относительно особенностей своего случая, прочитайте сам документ или обратитесь за консультацией к юристу.

Технические вопросы

Второе издание

Признаться, раньше я был немного циничен и полагал, что второе издание пишут, чтобы подгадить тем, кто собирается продать прочитанный экземпляр первого издания. Но это конкретное второе издание было бы невозможно без выхода первого и не могло бы выйти раньше (да и большинство читателей все равно читает электронные версии).

Самое крупное добавление, по сравнению с первым изданием, – глава о параллельных потоках, иначе говоря, о распараллеливании. Основное внимание в ней уделено библиотеке OpenMP и атомарным переменным и структурам. OpenMP не является частью стандарта C, но входит в экосистему C, и потому ее рассмотрение в этой книге вполне уместно. Атомарные переменные были добавлены в версию стандарта, опубликованную в декабре 2011 года, с этого момента до выхода первого издания даже года не прошло, поэтому в то время ни один компилятор их не поддерживал. Но с тех пор прогресс не стоял на месте, и я смог написать эту главу, опираясь как на изложенную в стандарте теорию, так и на реальный протестированный код (см. главу 12).

Первое издание почтили вниманием некоторые исключительно педантичные читатели. Они заметили все огрехи, которые можно было истолковать как ошибки, – от написанной мной глупости по поводу дефисов в командной строке до неправильно построенных предложений. Ничто в мире не совершенно, но благодаря конструктивным отзывам читателей книга стала гораздо точнее и полезнее.

Перечислю прочие дополнения к первому изданию.

- В приложении А приведен краткий справочник по языку С для читателей, имеющих опыт программирования на других языках. Мне не хотелось включать его в первое издание, потому что учебных пособий по С и так достаточно, но, как выяснилось, с ним книга стала полезнее.
- Откликаясь на часто высказываемое пожелание, я существенно расширил материал по работе с отладчиком (см. раздел «Работа с отладчиком»).
- В первом издании был раздел о том, как писать функции с переменным числом аргументов, так чтобы оба вызова `sum(1, 2.2)` и `sum(1, 2.2, 3, 8, 16)` были правильны. Но что, если требуется передать несколько списков, например вычислить скалярное произведение двух векторов произвольной длины: `dot((2, 4), (-1, 1))` и `dot((2, 4, 8, 16), (-1, 1, -1, 1))` (см. раздел «Несколько списков»)?
- Я переписал главу 11 о расширении объектов путем добавления новых функций. Основное дополнение – реализация виртуальных таблиц.
- Я немного дополнил материал о препроцессоре, уделив внимание трудному вопросу о тестовых макросах, включая и мимолетное упоминание ключевого слова `_Static_assert`.
- Я остался верен данному самому себе обещанию не включать в эту книгу справочный материал по регулярным выражениям (потому что в сети и в других книгах в этом нет недостатка). Но все же добавил демонстрационный пример в разделе «Разбор регулярных выражений», посвященном использованию определенных в POSIX функций для работы с регулярными выражениями, которые, по сравнению с другими языками, являются довольно низкоуровневыми.
- Обсуждение работы со строками в первом издании во многом опиралось на функцию `asprintf`, которая похожа на `sprintf`, но автоматически выделяет необходимое количество памяти перед записью в нее строки. Существует версия этой функции в дистрибутиве GNU, но многие читатели не могут ей воспользоваться из-за условий лицензии, поэтому в примере 9.3 в этом издании я показал, как написать такую функцию, пользуясь только стандартными конструкциями С.
- Один из серьезных вопросов, обсуждаемых в главе 7, – тот факт, что детальное управление числовыми типами может приводить к проблемам, поэтому в первом издании я не упомянул о десятках новых числовых типов, появившихся в стандарте C99, например: `int_least32_t`, `uint_fast64_t` и т. д. (C99 §7.18; C11 §7.20). Несколько читателей настойчиво просили меня отметить хотя бы наиболее полезные типы, например `intptr_t` и `intmax_t`, что я и делаю в подходящем месте.

Графические выделения

В книге применяются следующие графические выделения:

Курсив
Новые термины, URL-адреса, адреса электронной почты, имена и пути к файлам. Многие термины определены в глоссарии в конце книги.

Моноширинный
: Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Моноширинный курсив
: Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.


Так обозначается совет, рекомендация или замечание общего характера.


Так обозначаются упражнения, призванные помочь в освоении.


Так обозначается предупреждение или предостережение.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Примеры кода можно скачать с сайта **https://github.com/b-k/21st-Century-Examples**.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «21st Century C by Ben Klemens (O'Reilly). Copyright 2013 Ben Klemens, 978-1-449-32714-9».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу **permissions@oreilly.com**.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: **http://oreil.ly/21st_century_c**.

Замечания и вопросы технического характера следует отправлять по адресу **bookquestions@oreilly.com**.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу **http://www.oreilly.com**.

Читайте нас на Facebook: **http://facebook.com/oreilly**.

Следите за нашей лентой в Twitter: **http://twitter.com/oreillymedia**.

Смотрите нас на YouTube: **http://www.youtube.com/oreillymedia**.

Благодарности

- **Нора Альберт**: общая поддержка, подопытный кролик.
- **Брюс Филдс**, **Дэйв Китабян**, **Сара Вейссман**: скрупулезное рецензирование.
- **Патрик Холл**: знание Unicode.
- **Натан Джепсон**, **Эллисон Макдональд**, **Рейчел Румелиотис**, **Шон Уоллес**: редактирование.
- **Андреас Клейн**: указание на ценность типа `intptr_t`.
- **Роландо Родригес**: тестирование, любознательное использование, вдумчивое исследование.
- **Рейчел Стили**: производство.
- **Ульрик Свердруп**: указание на то, как использовать позиционные инициализаторы для задания значений по умолчанию.

Часть I

Окружение

В дикой природе за пределами ухоженных садиков скриптовых языков в изобилии водятся инструменты, способные решить пугающие проблемы C, но на них придется поохотиться. Действительно *придется*: многие из них абсолютно необходимы, если вы хотите писать код, не мучаясь. Если вы не пользуетесь отладчиком (автономным или в составе IDE), то взваливаете на себя совершенно ненужные трудности.

Существует также немало библиотек, которые только и ждут, чтобы вы воспользовались ими в своей программе и занимались решением поставленной задачи, не тратя времени на новую реализацию связанных списков, анализаторов и других вспомогательных средств. Компоновка программы с внешними библиотеками должна быть максимально простой.

Ниже приводится обзор первой части.

В главе 1 рассматривается настройка минимально необходимого окружения, в том числе использование менеджера пакетов для установки всех требуемых инструментов. Это подготовка к содержательной части – компиляции и компоновке своих программ со сторонними библиотеками. Сам процесс хорошо стандартизирован и предполагает использование немногих переменных окружения и рецептов.

В главе 2 мы познакомимся с инструментами для отладки, документирования и тестирования, потому что нельзя же назвать хорошим код, который не отлажен, не документирован и не протестирован!

Глава 3 посвящена Autotools – системе создания пакета для распространения программы. Но этим мы не ограничимся, а рассмотрим также написание скриптов оболочки и файлов makefile.

Ничто так не осложняет жизнь, как другие люди. Поэтому в главе 4 мы рассмотрим Git – систему учета слегка различающихся версий проекта на вашем диске и на дисках ваших коллег, которая елико возможно упрощает объединение версий.

Неотъемлемой частью современного окружения C являются другие языки, потому что у многих из них имеется программный интерфейс к C. В главе 5 будут приведены общие замечания касательно использования такого интерфейса с развернутым примером для языка Python.



Глава 1. Настраиваем среду для компиляции

Гляди, милочка, как я пользуюсь техникой.
— Игги Поп «Search and Destroy»

Одной лишь стандартной библиотеки С недостаточно для серьезной работы.

Поэтому экосистема С вышла за рамки стандарта, а это значит, что если вы собираетесь решать задачи посложнее упражнений в учебниках, то необходимо знать, как вызывать функции из распространенных, но не описанных в стандарте ISO библиотек. Чтобы работать с XML-документом, JPEG-изображением или с TIFF-файлом, нужны соответственно библиотеки libxml, libjpeg или libtiff. Все они свободно доступны, но частью стандарта не являются. Увы, на этом месте большинство учебников ставят многоточие и предоставляют вам искать выход самостоятельно, именно поэтому от хулителей С можно услышать такое внутренне противоречивое высказывание: *«С уже 40 лет, поэтому на нем нужно все писать с нуля»*. Да они просто так и не дали себе труда понять, как компоновать программу с библиотеками.

Вот план этой главы.

- Настроить необходимые инструменты. Сейчас это гораздо проще, чем в стародавние времена, когда приходилось охотиться за каждым компонентом. Подготовить полную систему сборки со всеми бантиками можно за 10, от силы 15 минут (плюс время скачивания всего добра).
- Как компилировать программу на С. Понимаю, вы знаете, как это делается, но нам нужна конфигурация, к которой можно подключить библиотеки и указать, где их искать; одна лишь команда `cc myfile.c` мало что даст. *Make* – пожалуй, простейшая система компиляции программ, поэтому при обсуждении мы возьмем ее за образец. Я покажу минимальный makefile, в котором вполне достаточно места для развития.
- Любая используемая нами система опирается на несколько переменных окружения, поэтому мы поговорим о том, для чего они нужны и как задаются. После того как вся инфраструктура компиляции подготовлена, добавление новых библиотек сводится просто к изменению уже имеющихся переменных.

- В качестве бонуса мы воспользуемся всем проделанным к этому моменту, чтобы настроить еще более простую систему компиляции, которая позволяет копировать код прямо в командную строку.

Отдельное замечание для работающих с IDE: пусть даже вы не имеете дела с make, но эта глава для вас небесполезна, потому что каждому рецепту, который исполняет make при компиляции программы, соответствует аналогичный рецепт в IDE. Если вы знаете, что делает make, то сможете без труда настроить параметры своей IDE.

Работа с менеджером пакетов

Не пользуясь менеджером пакетов, вы много теряете.

Я завел разговор о менеджерах пакетов по нескольким причинам. Во-первых, не исключено, что у кого-то не установлены даже базовые средства. Для них я включил этот раздел, поскольку эти инструменты необходимы, и как можно скорее. Хороший менеджер пакетов позволит очень быстро установить полную подсистему POSIX, компиляторы для всех языков, о которых вы слышали и не слышали, вполне достойный набор игр, обычные средства повышения продуктивности труда, несколько сотен написанных на C библиотек и т. д.

Во-вторых, для программистов на C менеджер пакетов – это тот важнейший компонент, с помощью которого мы получаем необходимые для работы библиотеки.

В-третьих, если вам доведется перейти из категории скачивающих пакеты в категорию создателей пакетов, то эта книга доведет вас до середины пути – научит, как подготовить пакет для автоматической установки. Впоследствии, если администратор репозитория пакетов захочет включить ваш код в репозиторий, у него не будет сложностей с построением окончательного пакета.

Если вы работаете в Linux, то менеджер пакетов уже использовался на стадии конфигурирования компьютера, поэтому вы видели, сколь простым может быть процесс получения программного обеспечения. А для пользователей Windows я подробно рассмотрю систему Cygwin. У пользователей Mac есть несколько вариантов, например Fink и Macports. Все они опираются на предлагаемую Apple систему Xcode, которая доступна бесплатно на установочном диске ОС, в каталоге допускающих установку программ, в магазине App Store или с помощью регистрации в качестве разработчика Apple (в зависимости от даты выпуска вашего Mac'a).

Какие пакеты вам понадобятся? Ниже приведен краткий ликбез по основам разработки на C. Разные системы организованы по-разному: состав дистрибутивов разнится, по умолчанию устанавливается то одно, то другое, иногда пакеты имеют странные имена. Если сомневаетесь относительно какого-то пакета, ставьте его, потому что прошли те дни, когда установка лишнего могла вызвать замедление или нестабильную работу системы. Однако вам может не хватить скорости сети (а то и места на диске), чтобы установить все имеющиеся в мире пакеты, поэтому нужно все же себя ограничивать. Если окажется, что вы о чем-то забыли,

всегда можно будет доставить пакет позже. Пакеты, которые совершенно точно необходимы:

- Компилятор. Обязательно установите gcc, возможно, имеется также Clang.
- gdb, отладчик.
- Valgrind, для проверки наличия ошибок работы с памятью.
- gprof, профилировщик.
- make, чтобы не вызывать компилятор напрямую.
- pkg-config, для поиска библиотек.
- Doxygen, для генерации документации.
- Текстовый редактор. Их сотни, выбирать есть из чего. Приведу несколько субъективных рекомендаций:
  - Emacs и vim любят крутые спецы. Emacs умеет все (буква *E* означает *extensible* – расширяемый), у vim возможностей поменьше, зато он очень удобен для тех, кто печатает вслепую. Если вы предчувствуете, что на работу с редактором придется потратить сотни часов, то имеет смысл освоить хотя бы один из этой парочки.
  - Kate – дружелюбный и симпатичный, предлагает неплохой набор функций, полезных программистам, например подсветку синтаксиса.
  - На крайний случай попробуйте nano – простой, как правда, редактор с текстовым интерфейсом, который, следовательно, работает даже тогда, когда графического интерфейса нет.
- Если вы поклонник IDE, выберите какую-нибудь одну – или несколько. Здесь выбор также велик, дам несколько советов.
  - Anjuta: входит в семейство GNOME. Дружит с Glade, конструктором графических интерфейсов GNOME.
  - KDevelop: входит в семейство KDE.
  - Code::blocks: относительно простая среда, работает в Windows.
  - Eclipse: автомобиль представительского класса, с кучей подставок для чашек и дополнительных кнопочек. Кстати, кросс-платформенный.

В последующих главах я расскажу и об инструментах для более тяжелых работ:

- Autotools: Autoconf, Automake, libtool;
- Git;
- альтернативные оболочки, в том числе zsh.

Ну и, разумеется, написанные на C библиотеки, которые избавят вас от необходимости изобретать колесо (или, если употребить более точную метафору, паровоз). Возможно, вам понадобится многое другое, но ниже перечислены библиотеки, которые используются в этой книге:

- libcURL;
- libGlib;
- libGSL;
- libSQLite3;
- libXML2.

По поводу именования библиотечных пакетов нет единодушия, и вам придется самостоятельно разбираться в том, как менеджер пакетов предпочитает разбивать

единую библиотеку на части. Как правило, существует один пакет для пользователей и другой для разработчиков, которые собираются использовать библиотеку в своих проектах, поэтому ставьте как базовый пакет, так и пакет с именем, оканчивающимся на `-dev` или `-devel`. В некоторых системах документация поставляется в виде отдельного пакета. А бывает так, что отладочные символы нужно скачивать отдельно – тогда при первой попытке отладить код, для которого нет отладочных символов, gdb подскажет, что сделать.

Если вы работаете в POSIX-системе, то после установки всего вышеперечисленного получите полную систему разработки и сможете приступить к кодированию. Для пользователей Windows мы сделаем небольшое отступление, в котором объясним, как процедура установки взаимодействует с операционной системой.

Компиляция программ на C в Windows

В большинстве систем C – это центральный, «привилегированный» язык, прихотям которого служат все прочие инструменты; в Windows C почему-то оказался в загоне. Поэтому я потрачу немного времени, чтобы объяснить, как обустроить машину под управлением Windows для программирования на C. Если вы в данный отрезок времени не пишете в Windows, то переходите сразу к разделу «Как пройти в библиотеку?».

POSIX в Windows

Поскольку C и Unix развивались параллельно, трудно говорить об одном и не говорить о другом. Полагаю, проще всего начать со стандарта POSIX. К тому же программистам, пытающимся откомпилировать в Windows код, написанный в какой-то другой системе, этот путь покажется самым естественным.

Насколько мне известно, все, что так или иначе связано с файловыми системами, можно разбить на два слабо перекрывающихся класса:

- POSIX-совместимые системы;
- семейство операционных систем Windows.

Совместимость с POSIX не означает, что система должна выглядеть и работать, как Unix. Например, типичный пользователь Mac понятия не имеет, что использует стандартную систему BSD с красивым фасадом, но знающие люди могут перейти в папку Accessories → Utilities, открыть программу Terminal и выполнять милые их душе команды `ls`, `grep` или `make`.

И, кстати говоря, я сомневаюсь, что много найдется систем, на все 100% отвечающих требованиям стандарта (например, о наличии компилятора Fortran 77). Для наших целей достаточно POSIX-оболочки, дюжины утилит (`sed`, `grep`, `make`, ...), компилятора C99 и дополнений к стандартной библиотеке C, в частности `fork` и `iconv`. Их можно доустановить в качестве довеска к основной системе. Менеджер пакетов, внутренние скрипты, Autotools и почти все прочие средства для написания переносимого кода в той или иной степени опираются на эти инструменты, так что даже если вы и не собираетесь глазеть весь день на командную строку, установить их было бы полезно.

В серверных ОС и полных изданиях Windows 7 Майкрософт предлагает то, что раньше называлось INTERIX, а теперь «подсистема для приложений на базе Unix» (SUA). Она включает стандартные системные вызовы POSIX, оболочку Корна и компилятор gcc. По умолчанию эта подсистема обычно не устанавливается, но может быть установлена в качестве дополнительного компонента. Однако для других современных изданий Windows SUA недоступна, нет ее и в Windows 8, поэтому нельзя предполагать наличие подсистемы POSIX во всех операционных системах Майкрософт.

А раз так, то переходим к Cygwin.

Если бы нужно было создать Cygwin с нуля, то следовало бы действовать по такому плану.

1. Написать на C библиотеку для Windows, которая предоставляет все функции POSIX. Она должна будет сгладить несоответствия между Windows и POSIX, например тот факт, что в Windows диски обозначаются буквами (*C:*), а в POSIX файловая система однородна. В данном случае диск *C:* получает псевдоним */cygdrive/c*, диск *D:* – псевдоним */cygdrive/d* и т. д.
2. Теперь, когда мы можем откомпилировать любую POSIX-совместимую программу, скомпоновав ее с нашей библиотекой, нужно построить Windows-версии программ `ls`, `bash`, `grep`, `make`, `gcc`, `X`, `rxvt`, `libglib`, `perl`, `python` и т. д.
3. Построив сотни программ и библиотек, нужно настроить менеджер пакетов, который позволял бы пользователям выбирать, что устанавливать.

Пользователю Cygwin нужно всего лишь скачать менеджер пакетов по ссылке на программу установки на сайте Cygwin по адресу *http://cygwin.com* и выбрать нужные пакеты. Безусловно, вам понадобится все вышеперечисленное, а также приличный терминал (попробуйте mintty или установите подсистему X и пользуйтесь xterm, то и другое куда лучше стандартной программы `cmd.exe`, включенной в Windows), но вообще-то, как вы увидите, к вашим услугам практически все богатство системы разработки. Вот теперь можно приступать к компиляции кода.

Компиляция программ на C при наличии подсистемы POSIX

Майкрософт поставляет компилятор C++, входящий в состав Visual Studio, а у него есть режим совместимости с C89 (обычно он называется *ANSI C*, хотя в настоящее время стандартом ANSI является C11). И это единственное средство компиляции программ на C, предлагаемое Майкрософт. Представители компании ясно дали понять, что ожидать чего-нибудь, кроме поддержки нескольких возможностей, описанных в C99 (не говоря уже о поддержке C11), не следует. Visual Studio – единственный из основных компиляторов, так и застрявший на C89, поэтому придется искать альтернативы в других местах.

Разумеется, в Cygwin есть gcc, так что если вы, следуя моим указаниям, установили Cygwin, то все окружение сборки уже готово.

По умолчанию программы, скомпилированные в среде Cygwin, зависят от библиотеки функций POSIX *cygwin1.dll* (даже если программа нигде не обращается

к функциям POSIX). Если запустить программу на машине, где Cygwin установлена, то проблем не возникнет. Щелчок по исполняемому файлу приведет к его запуску, поскольку система найдет DLL-библиотеку Cygwin. Программы, скомпилированные в среде Cygwin, можно запускать и на машине, где Cygwin не стоит, нужно только включить в дистрибутив файл *cygwin1.dll*. На моей машине он находится в каталоге *(путь к cygwin)/bin/cygwin1.dll*. Файл *cygwin1.dll* поставляется на условиях GPL-подобной лицензии (см. врезку «Юридические вопросы» выше), то есть если вы распространяете его отдельно от Cygwin, то обязаны опубликовать исходный код своей программы[1].

Если это оказывается проблемой, то вам придется найти способ переписать программу, так чтобы она не зависела от *cygwin1.dll*, а это означает, что нужно отказаться от всех специфичных для POSIX функций (например, `fork` или `popen`) и использовать комплект MinGW, как описано ниже. С помощью утилиты `cygcheck` можно узнать, от каких DLL зависит ваша программа, и таким образом проверить, компонуется ваш исполняемый файл с *cygwin1.dll* или нет.



Узнать, от каких еще библиотек зависит данная динамическая библиотека, можно следующим образом:

- Cygwin: `cygcheck libxx.dll`
- Linux: `ldd libxx.so`
- Mac: `otool -L libxx.dylib`

Компиляция программ на С в отсутствие подсистемы POSIX

Если вашей программе не нужны функции POSIX, то можно воспользоваться комплектом MinGW (Minimalist GNU for Windows), в котором есть стандартный компилятор C и несколько простых инструментов. Дополнением к MinGW служит комплект MSYS, содержащий оболочку и другие полезные утилиты.

Вы можете пользоваться POSIX-оболочкой, входящей в состав MSYS (там же найдется терминал mintty или RXVT, в котором эта оболочка будет исполняться), или вообще отказаться от командной строки и попробовать интегрированную среду Code::blocks, в которой MinGW применяется для компиляции в Windows. Eclipse – гораздо более развитая IDE, которую тоже можно настроить под MinGW, хотя это требует немного больше работы.

Если же вам приятнее работать с командной оболочкой POSIX, настройте Cygwin, установите пакеты, содержащие версию gcc из комплекта MinGW, и пользуйтесь этим компилятором, а не включенной в Cygwin по умолчанию версией gcc, которая компонует вашу программу с библиотекой POSIX.

Если вы раньше не сталкивались с Autotools, то встречи осталось ждать недолго. Признаком пакета, построенного с помощью Autotools, является процедура установки, состоящая из трех шагов: `./configure && make && make install`. MSYS предоставляет все необходимое, чтобы такие пакеты заработали (с большой вероятностью).

---

[1] Проект Cygwin сопровождает компания Red Hat, Inc., которая дает возможность купить право не распространять исходный код, как того требует GPL.

Ну а если вы скачали пакеты из командной строки Cygwin и хотите собрать их самостоятельно, то для того чтобы при сборке пакета использовался компилятор Mingw32, порождающий независимый от POSIX код, нужно следующим образом модифицировать первый шаг:

```
./configure --host=ming32
```

Затем выполните `make; make install`, как обычно.

Результатом компиляции пакета с помощью MinGW, все равно, из командной строки или средствами Autotools, является машинный код для Windows. Поскольку MinGW ничего не знает о *cygwin1.dll*, а ваша программа к функциям POSIX не обращается, то получится честная Windows-программа, и никто не сможет сказать, что она была откомпилирована в окружении POSIX.

Однако в настоящее время для MinGW очень мало готовых откомпилированных библиотек[1]. Если вы хотите освободиться от *cygwin1.dll*, то не сможете воспользоваться версией *libglib.dll*, которая идет с Cygwin. Вам придется перекомпилировать GLib из исходных текстов в нормальную Windows DLL, однако GLib зависит от библиотеки gettext, разработанной GNU для интернационализации, поэтому сначала придется собрать ее. Современный код зависит от множества библиотек, поэтому вы, скорее всего, потратите уйму времени на действия, которые в других системах требуют всего одного обращения к менеджеру пакетов. Вот мы и вернулись к тому положению вещей, которое заставляет говорить о том, что C, мол, уже 40 лет, поэтому все нужно писать с нуля.

Так что подводные камни есть. Майкрософт ушла от разговора, оставив другим реализовывать постпанковский компилятор C и окружение для него. Cygwin приняла вызов и предоставляет полный менеджер пакетов с достаточным набором библиотек, который позволяет вам решить поставленную задачу целиком или хотя бы частично. Однако этот набор опирается на стандарт POSIX и библиотеку Cygwin. Если это оказывается проблемой, то для создания окружения и сборки необходимых библиотек вам придется проделать дополнительную работу.

Как пройти в библиотеку?

Итак, у вас есть компилятор, комплект инструментов, определенный в стандарте POSIX, и менеджер пакетов, позволяющий установить сотни библиотек. Теперь займемся вопросом о том, как воспользоваться этими библиотеками при компиляции своей программы.

---

[1] Хотя в комплекте MinGW есть менеджер пакетов, который устанавливает основные средства системы и предлагает кое-какие библиотеки (большая их часть нужна самому MinGW), эта кучка заранее откомпилированных библиотек – ничто по сравнению с сотнями пакетов, предлагаемых типичным менеджером пакетов. На самом деле менеджер пакетов на моей Linux-машине предлагает даже больше библиотек, откомпилированных для MinGW, чем менеджер пакетов из MinGW. Но это на момент написания книги; не исключено, что когда вы будете ее читать, такие же энтузиасты, как вы сами, положат в репозиторий MinGW дополнительные пакеты.

Начать нужно с вызова компилятора из командной строки, а соответствующая команда очень быстро становится несуразно длинной. Однако есть три (иногда три с половиной) простых шага, облегчающих жизнь.

1. Задать переменную, содержащую флаги компилятора.
2. Задать переменную, содержащую список библиотек, с которыми компонуется программа. Половина шага связана с тем, что иногда нужно задавать одну переменную для компоновки на этапе компиляции, а иногда две – для компоновки на этапе компиляции и на этапе выполнения.
3. Настроить систему, так чтобы эти переменные учитывались при компиляции.

Чтобы воспользоваться библиотекой, необходимо дважды сообщить, что вы собираетесь импортировать из нее функции: один раз – компилятору, второй – компоновщику. Если библиотека находится в стандартном месте, то первое объявление производится с помощью директивы `#include` в тексте программы, а второе – с помощью флага `-l` при вызове компилятора.

В примере 1.1 показана простая программа, которая выполняет интересное математическое вычисление (по крайней мере, интересное для меня; если для вас статистическая терминология – абракадабра, ничего страшного). Описанная в стандарте C99 ***функция ошибок***, `erf(x)`, тесно связана с интегралом от 0 до x функции нормального распределения со средним 0 и стандартным отклонением $\sqrt{2}$. В данном случае мы используем функцию `erf`, чтобы вычислить площадь популярной у статистиков области (95-процентный доверительный интервал для критерия проверки гипотезы о нормальном распределении). Назовем этот файл *erf.c*.

Пример 1.1 ❖ Однострочная программа с вызовом функции из стандартной библиотеки (erf.c)

```
#include <math.h> //erf, sqrt
#include <stdio.h> //printf

int main(){
  printf("The integral of a Normal(0, 1) distribution "
         "between -1.96 and 1.96 is:%g\n", erf(1.96*sqrt(1/2.)));
}
```

Директивы `#include` должны быть вам знакомы. Компилятор вставляет вместо них содержимое файлов *math.h* и *stdio.h* соответственно, а следовательно, и объявления функций `printf`, `erf` и `sqrt`. Объявление в *math.h* ничего не говорит о том, что делает функция *erf*, известно лишь, что она принимает параметр типа `double` и возвращает значение типа *double*. Этой информации компилятору достаточно для проверки правильности использования и создания объектного файла, в котором оставлено сообщение компьютеру: когда дойдешь до этого места, найди функцию `erf` и подставь сюда возвращаемое ей значение.

Отреагировать на это сообщение должен компоновщик, который и производит поиск функции `erf` в библиотеке, находящейся где-то на диске.

Математические функции, объявленные в заголовке `math.h`, живут в отдельной библиотеке, и компоновщику необходимо сказать об этом с помощью флага `-lm`. Здесь `-l` означает, что необходимо прикомпоновать библиотеку, имя которой в дан-

ном случае состоит всего из одной буквы m. Функцию printf мы получаем задаром, потому что при вызове компилятора неявно подразумевается, что в самом конце команды указан флаг -lc, требующий от компоновщика добавить стандартную библиотеку libc. Позже мы увидим, что библиотека GLib 2.0 компонуется с помощью флага -lglib-2.0, библиотека GNU Scientific Library – с помощью флага -lgsl и т. д.

Таким образом, если файл называется *erf.c*, то полная команда вызова компилятора gcc с несколькими дополнительными флагами, которые мы обсудим ниже, выглядит так:

```
gcc erf.c -o erf -lm -g -Wall -O3 -std=gnu11
```

Итак, с помощью директивы #include в тексте программы мы попросили компилятор включить математические функции, а с помощью флага -lm в командной строке попросили компоновщик прикомпоновать математическую библиотеку.

Флаг -o задает имя выходного файла, без него мы по умолчанию получили бы исполняемый файл с именем a.out.

Несколько моих любимых флагов

Как вы скоро увидите, я всегда указываю несколько флагов компилятора и вам советую поступать так же.

- -g – включить отладочные символы. Без этого флага отладчик не покажет вам имена переменных и функций. Наличие отладочных символов не замедляет работу программы, а то, что файл станет на килобайт больше, несущественно, поэтому нет никаких причин не пользоваться ими. Этот флаг понимают gcc, Clang и icc (Intel C Compiler).
- -std=gnu11 – этот флаг понимают clang и gcc, он означает, что компилятор должен разрешать код, совместимый со стандартами C11 и POSIX (а также некоторые расширения GNU). На момент написания данной книги clang по умолчанию подразумевает стандарт C99, а gcc – стандарт C89. Если у вас стоит версия gcc, clang или icc, предшествующая выходу стандарта C11, задавайте флаг -std=gnu99, чтобы работать на уровне C99. Стандарт POSIX требует, чтобы в системе присутствовал компилятор c99, поэтому не содержащая версии стандарта командная строка для компиляции кода, совместимого с C99, имеет вид:

  ```
  c99 erf.c -o erf -lm -g -Wall -O3
  ```

В показанных ниже файлах makefile я достигаю того же эффекта, установив переменную CC=c99.



В зависимости от года выпуска Mac c99 может быть специально подправленной версией gcc, а это, возможно, не то, что вам нужно. Если ваша версия c99 падает при задании флага -Wall или такой программы вообще нет, сделайте свою собственную версию. В файл инициализации оболочки (скорее всего, .bashrc) добавьте псевдоним

```
alias c99="gcc --std=c99"
```

или

```
alias c99="clang"
```

как вам больше нравится.

- `-O3` – задает уровень оптимизации 3, при котором компилятор делает все возможное для построения более быстрого кода. Если при работе с отладчиком оказывается, что из-за оптимизации исчезло слишком много переменных и стало трудно следить за тем, что происходит, то задайте флаг `-O0`. Ниже мы увидим стандартный способ сделать это с помощью переменной `CFLAGS`. Флаг понимают gcc, Clang и icc.
- `-Wall` – выводить все предупреждения компилятора. Годится для gcc, Clang и icc. В случае icc предпочтительнее флаг `-w1`, который задает режим вывода предупреждений компилятора, опуская замечания.



Всегда включайте вывод предупреждений. Даже если вы думаете, что знаете стандарт C вдоль и поперек, компилятор все равно знает лучше. В старых учебниках C целые страницы заполнены наставлениями не путать = и == и проверять, что каждая переменная инициализируется перед использованием. Будучи автором более современного учебника, я поступлю проще, сведя все наставления к одному: обязательно включайте предупреждения компилятора. Если компилятор рекомендует что-то изменить, не спорьте с ним и не откладывайте исправление на потом. Сделайте все, чтобы (1) понять, с чем связано предупреждение, и (2) исправить код, так чтобы он компилировался вообще без ошибок и предупреждений. Невразумительность сообщений компилятора давно стала притчей во языцех, поэтому если с пунктом (1) возникают сложности, скопируйте текст предупреждения в поисковик – вы узнаете, сколько народу до вас зашло в тупик, столкнувшись с ним. Возможно, вы захотите задать флаг `-Werror`, чтобы компилятор трактовал все предупреждения как ошибки.

Пути

На моем диске больше 700 000 файлов, из которых один содержит объявления функций `sqrt` и `erf`, а другой – объектный код откомпилированных функций. (Чтобы получить грубую оценку количества файлов в POSIX-совместимой системе, выполните команду `find / -type f | wc -l`.) Компилятор должен знать, в каких каталогах искать заголовки и объектные файлы, и эта проблема еще осложняется, когда мы пользуемся библиотеками, не описанными в стандарте C.

В типичной системе библиотеки могут находиться, по крайней мере, в трех местах.

- Поставщик операционной системы определяет один или два стандартных каталога, где находятся поставляемые им библиотеки.
- Может существовать каталог, в который системный администратор устанавливает пакеты, которые не должны быть перезаписаны при очередном обновлении ОС поставщиком. Там же могут находиться специальные версии библиотек, подправленные администратором.
- У обычных пользователей обычно нет прав для записи в эти места, поэтому должна быть возможность использовать библиотеки, находящиеся в их домашних каталогах.

Что касается стандартных мест, то тут обычно проблем не возникает, компилятор знает, где искать стандартную библиотеку C и все установленное вместе с ней. В стандарте POSIX такие каталоги называются «обычными местами».

Но про все остальное компилятору нужно сказать явно. И тут мы сталкиваемся с византийским вероломством: нет никакого стандартного способа поиска библио-

тек в нестандартных местах, и эта неприятность занимает высокое место в списке вещей, вызывающих раздражение при работе с C. Но есть и хорошие новости: раз компилятор знает, где находятся обычные места, то распространители библиотек стараются ставить их именно туда, чтобы пользователям не приходилось задавать путь вручную. Еще одна хорошая новость заключается в том, что есть несколько инструментов, помогающих в задании путей. И последнее приятное известие: один раз поняв, где расположены нестандартные места, вы можете прописать их в переменной оболочки или makefile и больше об этом не думать.

Предположим, что на вашем компьютере имеется библиотека Libuseful и вы знаете, что различные относящиеся к ней файлы помещены в каталог */usr/local/*, который официально предназначен для локальных библиотек, устанавливаемых администратором. Вы уже включили в код директиву `#include <useful.h>`, и теперь нужно следующим образом сформировать командную строку:

```
gcc -I/usr/local/include use_useful.c -o use_useful -L/usr/local/lib -luseful
```

Здесь:

- Флаг `-I` добавляет путь в список каталогов, где компилятор ищет файлы, включаемые с помощью директивы `#include`.
- Флаг `-L` добавляет путь в список каталогов, где ищутся библиотеки.
- Порядок важен. Если файл *specific.o* зависит от библиотеки Libbroad, а библиотека Libbroad зависит от Libgeneral, то перечислять их нужно в следующем порядке:

  ```
  gcc specific.o -lbroad -lgeneral
  ```

  При любом другом порядке, например `gcc -lbroad -lgeneral specific.o`, компоновка, скорее всего, завершится неудачно. Можете представлять этот процесс следующим образом: компоновщик сначала видит первый файл, `specific.o`, и составляет список неразрешенных ссылок на функции, структуры и имена переменных. Затем он переходит к указанной далее библиотеке, `-lbroad`, и ищет в ней недостающие элементы, попутно добавляя в список новые неразрешенные ссылки. Наконец, элементы, оказавшиеся в списке отсутствующих, ищутся в библиотеке `-lgeneral`. Если по завершении этого процесса (не забудем о неявно добавляемой в конце библиотеке `-lc`) какие-то имена остались неразрешенными, то компоновщик завершает работу и выводит эти имена на экран.

Но вернемся к проблеме местонахождения: где та библиотека, с которой вы хотите скомпоновать программу? Если она была установлена тем самым менеджером пакетов, который использовался для установки всей системы, то, скорее всего, находится в одном из обычных мест и беспокоиться не о чем.

Возможно, вы представляете, где могут находиться локальные библиотеки, например в каталогах */usr/local*, */sw* или */opt*. Разумеется, к вашим услугам различные средства поиска на диске, в частности определенная в POSIX утилита find. Так, команда

```
find /usr -name 'libuseful*'
```

ищет в каталоге */usr* файлы с именами, начинающимися с *libuseful*. Если разделяемый объектный файл библиотеки Libuseful найден в каталоге */some/path/lib*, то ее заголовки почти наверняка находятся в каталоге */some/path/include*.

Но охота за библиотеками по всему диску мало кому нравится, поэтому утилита pkg-config решает эту проблему, сохраняя в репозитории флаги и места, которые сами пакеты объявляют необходимыми для компиляции. Наберите в командной строке pkg-config; если появится сообщение с просьбой задать имена пакетов, значит, все нормально: у вас есть pkg-config, и ей можно воспользоваться для изысканий. Например, на моей машине следующие две команды:

```
pkg-config --libs gsl libxml-2.0
pkg-config --cflags gsl libxml-2.0
```

печатают соответственно:

```
-lgsl -lgslcblas -lm -lxml2
-I/usr/include/libxml2
```

Это и есть те флаги, которые нужно задать при компиляции GSL и LibXML2. Флаг -l показывает, что библиотека GNU Scientific Library зависит от библиотеки Basic Linear Algebra Subprograms (BLAS), а та, в свою очередь, зависит от стандартной математической библиотеки. Вроде бы все библиотеки находятся в обычных местах, потому что флагов -L не наблюдается, однако флаг -I определяет специальное место для заголовочных файлов LibXML2.

Вернемся к командной строке; у оболочки есть интересная особенность: если заключить команду в обратные апострофы, то вместо команды будет подставлен ее результат. Иначе говоря, если ввести команду:

```
gcc `pkg-config --cflags --libs gsl libxml-2.0` -o specific specific.c
```

то компилятор увидит:

```
gcc -I/usr/include/libxml2 -lgsl -lgslcblas -lm -lxml2 -o specific specific.c
```

Таким образом, pkg-config многое делает за нас, но в стандарте она не описана, поэтому нельзя ожидать, что она имеется на любой машине или что любая библиотека регистрируется в ее репозитории. Если у вас этой утилиты нет, то придется заняться самостоятельными исследованиями: почитать руководство по библиотеке или поискать на диске, как было показано выше.



Часто существуют специальные переменные окружения для путей, например CPATH или LIBRARY_PATH или C_INCLUDE_PATH. Их можно задать в своем файле .bashrc или в каком-нибудь другом пользовательском перечне переменных окружения. Впрочем, они безнадежно нестандартные – gcc в Linux и gcc в Mac используют разные переменные, а другой компилятор может понимать что-то совсем иное. Я считаю, что проще задавать эти пути для каждого проекта в отдельности в файле makefile или эквивалентном ему с помощью флагов -I и -L. Если вы все же предпочитаете вышеупомянутые переменные путей, то хотя бы посмотрите в конце страницы руководства по своему компилятору список поддерживаемых переменных.

Даже при наличии pkg-config возникает острая необходимость в каком-нибудь инструменте, который автоматически соберет все необходимое воедино. Каждый отдельный элемент понять несложно, но долгая механическая работа утомляет.

Компоновка во время выполнения

При компоновке со *статическими библиотеками* компилятор, по сути дела, копирует содержимое библиотеки в конечный исполняемый файл. Поэтому программа работает более-менее автономно. *Разделяемые библиотеки* прикомпоновываются к программе во время выполнения, то есть мы имеем ту же проблему поиска библиотек, что и на этапе компиляции. Хуже того, теперь с этой проблемой могут столкнуться *пользователи* нашей программы.

Если библиотека находится в одном из обычных мест, то жизнь прекрасна – во время выполнения система найдет библиотеку без труда. Если же ваша библиотека находится в нестандартном месте, то нужен какой-то способ изменить путь поиска библиотек во время выполнения. Вот какие есть варианты.

- Если пакет программы собирался с помощью Autotools, то Libtool знает, какие флаги добавить, и вам беспокоиться не о чем.
- Наиболее вероятная причина, по которой необходимо изменить путь поиска, состоит в том, что вы храните библиотеки в своем домашнем каталоге, потому что не имеете прав суперпользователя (или не хотите ими пользоваться). Если все свои библиотеки вы устанавливаете в каталог *libpath*, то нужно задать переменную окружения `LD_LIBRARY_PATH`. Обычно это делается в файле инициализации оболочки (`.bashrc`, `.zshrc` или еще каком-то) с помощью команды:

  ```
  export LD_LIBRARY_PATH=libpath:$LD_LIBRARY_PATH
  ```

  Можно услышать предостережения против бездумного использования переменной `LD_LIBRARY_PATH` (что, если кто-нибудь поместит в указанный в ней каталог вредоносную библиотеку с таким же именем, как у настоящей, а вы об этом не будете знать?), однако если все ваши библиотеки находятся в одном месте, то не так уж страшно добавить в путь один каталог, который вы будете тщательно контролировать.
- Если программа компилируется с помощью gcc, Clang или icc, то в предположении, что библиотека находится в *libpath*, добавьте строку

  ```
  LDADD=-Llibpath -Wl,-Rlibpath
  ```

  в соответствующий makefile. Флаг `-L` говорит компилятору, где искать библиотеки для разрешения символов, флаг `-Wl` передает следующие далее флаги компоновщику через gcc/Clang/icc, а компоновщик вставляет каталог, указанный во флаге `-R`, в путь поиска библиотек, просматриваемый во время выполнения. К сожалению, `pkg-config` обычно не знает о путях на этапе выполнения, поэтому эти флаги, возможно, придется добавлять вручную.

Работа с файлами makefile

Файл *makefile* кладет конец всем этим бесконечным настройкам. По существу, это организованный набор переменных и однострочных скриптов оболочки. Описанная в стандарте POSIX программа *make* читает команды и переменные из makefile,

после чего конструирует за нас длинные командные строки. Прочитав этот раздел, вы уже вряд ли станете когда-нибудь вызывать компилятор напрямую.

В разделе «Makefile, или сценарий оболочки» ниже я еще вернусь к файлам makefile, а пока приведу минимальный практически полезный makefile, который собирает простенькую программу, зависящую от одной библиотеки. Вот он целиком, всего шесть строчек:

```
P=program_name
OBJECTS=
CFLAGS = -g -Wall -O3
LDLIBS=
CC=c99

$(P): $(OBJECTS)
```

Как им пользоваться?

- Однократно: сохраните этот файл (под именем *makefile*) в том же каталоге, где находятся файлы с расширением *.c*. Если вы пользуетесь программой GNU Make, то можете писать имя с большой буквы (*Makefile*), чтобы выделить его среди прочих файлов. В первой строке укажите имя своей программы (`progname`, а не `progname.c`).
- При каждой компиляции: наберите `make`.



Ваша очередь. Вот пример знаменитой программы hello.c [Kernighan 1978, стр. 6], состоящей всего из двух строк:

```
#include <stdio.h>
int main(){ printf("Hello, world.\n"); }
```

Сохраните этот файл и показанный выше makefile в каком-нибудь каталоге и попробуйте выполнить описанные шаги для компиляции программы. Затем запустите ее.

Задание переменных

Скоро мы перейдем к вопросу о работе makefile, а пока отметим, что пять из шести строк – это присваивания значений переменным (многие сейчас пусты). Это означает, что следует несколько подробнее побеседовать о переменных окружения.



Исторически сложились два направления в разработке оболочек: первое основано преимущественно на оболочке Боурна, второе – на С-оболочке (C shell). В С-оболочке синтаксис задания переменных несколько отличается, например чтобы присвоить значение переменной `CFLAGS`, нужно написать `set CFLAGS="-g -Wall -O3"`. Но в стандарте POSIX описан только синтаксис, основанный на оболочке Боурна, поэтому он и используется в этой книге.

И в оболочке, и в программе `make` значение переменной обозначается знаком `$`, но в оболочке пишут `$var`, а в `make` имена переменных длиннее одного символа заключаются в скобки: `$(var)`. Поэтому при вычислении строки `$(P): $(OBJECTS)` в показанном выше makefile получится

```
program_name:
```

Есть несколько способов уведомить make о наличии переменной.

- Установить переменную в оболочке перед вызовом *make* и *экспортировать* ее. Это означает, что при запуске дочернего процесса оболочка поместит данную переменную в список передаваемых ему переменных окружения. Чтобы установить переменную CFLAGS в командной строке POSIX-совместимой оболочки, нужно написать:

  ```
  export CFLAGS='-g -Wall -O3'
  ```

  На своем домашнем компьютере я опускаю первую строку в файле makefile (P=program_name) и вместо этого задаю переменную на все время сеанса командой export P=program_name, что позволяет редактировать сам makefile не так часто.
- Команду экспорта можно поместить в файл инициализации оболочки, например .bashrc или .zshrc. Это означает, что при каждом входе в систему или при запуске новой оболочки эта переменная будет установлена и экспортирована. Если вы точно знаете, что CFLAGS всегда должна принимать одно и то же значение, можете установить ее в этом файле и забыть.
- Можно экспортировать переменную только для одной команды, поместив присваивание перед этой командой. Команда env выводит список установленных переменных окружения, поэтому, выполнив команду

  ```
  PANTS=kakhi env | grep PANTS
  ```

  вы увидите значение переменной PANTS. Именно поэтому оболочка не позволяет ставить пробелы до и после знака равенства: пробел позволяет отличить присваивание от команды.

  В этом случае переменная устанавливается и экспортируется только для команды в той же строке. Выполнив показанную выше команду, попробуйте запустить env | grep PANTS и убедитесь, что переменная PANTS больше не экспортируется.

  Никто не мешает задавать сразу несколько переменных:

  ```
  PANTS=kakhi PLANTS="ficus fern" env | grep 'P.*NTS'
  ```

  Этот прием, называемый в спецификации оболочки *простой командой*, подразумевает, что присваивание производится непосредственно перед записью самой команды. Это окажется существенно, когда мы перейдем к конструкциям оболочки, отличным от команд. Предложение

  ```
  VAR=val if [ -e afile ] ; then ./program_using_VAR ; fi
  ```

  завершится с маловразумительной синтаксической ошибкой. Правильно писать так:

  ```
  if [ -e afile ] ; then VAR=val ./program_using_VAR ; fi
  ```

- Как в показанном выше makefile, можно задать переменную в начале файла, в строках вида CFLAGS=... . В makefile разрешается оставлять пробелы до и после знака равенства.

- make позволяет задавать переменные в командной строке независимо от оболочки. Таким образом, следующие две строки почти эквивалентны:

```
make CFLAGS="-g -Wall"   Установить переменную в makefile.
CFLAGS="-g -Wall" make   Установить переменную окружения, видимую только make и ее потомкам.
```

Все эти способы, с точки зрения работы makefile, эквивалентны, за исключением одного нюанса: программы, вызываемые из make, будут видеть новые переменные окружения, но ничего не будут знать о переменных, установленных в самом makefile.

Переменные окружения в C

В программе на C переменную окружения можно получить с помощью функции getenv. Поскольку getenv очень проста, ее можно использовать для проверки эффекта задания разных значений в командной строке.

В примере 1.2 сообщение выводится на экран столько раз, сколько укажет пользователь. Текст сообщения задается в переменной окружения msg, а число повторений – в переменной reps. Обратите внимание, что в случае, когда getenv возвращает NULL (обычно это означает, что переменная окружения не задана), мы по умолчанию берем значения 10 и «Hello».

Пример 1.2 ❖ Переменные окружения позволяют быстро изменить параметры работы программы (getenv.c)

```
#include <stdlib.h>//getenv, atoi
#include <stdio.h> //printf

int main(){
char *repstext = getenv("reps");
int reps = repstext ? atoi(repstext) : 10;

char *msg = getenv("msg");

if (!msg) msg = "Hello.";
for (int i=0; i< reps; i++)
   printf("%s\n", msg);
}
```

Как и раньше, мы можем экспортировать переменные, так чтобы они действовали только в данной строке; это еще больше упрощает процесс передачи переменной программе.

```
reps=10 msg="Ha" ./getenv
msg="Ha" ./getenv
reps=20 msg=" " ./getenv
```

Выглядит это странновато, ведь естественно ожидать, что параметры программы указываются *после* имени, но бог с ней, со странностью, зато внутри самой программы нам почти ничего не пришлось делать, так что мы получили именованные параметры чуть ли не задаром.

Набравшись побольше опыта, вы сможете изучить описанную в POSIX функцию getopt или предлагаемую GNU функцию argp_parse, которые позволяют обрабатывать входные параметры обычным способом.

В make есть также несколько встроенных переменных. Ниже перечислены те из них (все описаны в POSIX), которые встретятся в рассматриваемых далее правилах.

`$@`

Полное имя целевого файла. Под *целевым* понимается файл, который предстоит создать, например *o*-файл, получающийся в результате компиляции *c*-файла, или программа, являющаяся результатом компоновки *o*-файлов.

`$*`

Имя целевого файла без суффикса. Так, если целевой файл называется *prog.o*, то `$*` содержит *prog*, а `$*.c` принимает значение *prog.c*.

`$<`

Имя файла, в ходе обработки которого создается текущий целевой файл. Создание файла *prog.o*, вероятно, обусловлено тем, что недавно изменился файл *prog.c*, поэтому `$<` будет равно *prog.c*.

Правила

Теперь перенесем внимание на процедуры, выполняемые при обработке файла makefile, а затем обсудим, как на этот процесс влияют переменные.

Помимо задания переменных, в makefile имеются такие части:

```
цель: зависимости
      скрипт
```

Если инициируется создание некоторой цели путем исполнения команды `make target`, то система проверяет зависимости. Если целью является файл, если эта цель зависит только от файлов и при этом цель новее (то есть изменена позже) своих зависимостей, то файл актуален, и ничего делать не нужно. В противном случае обработка цели откладывается, выполняются или генерируются ее зависимости, быть может, с помощью другой цели, а когда обработка всех зависимостей завершится, выполняются команды создания исходной цели.

Например, прежде чем стать книгой, этот текст был серией статей в блоге (**http://modelingwithdata.org**). У каждой статьи было две версии – HTML и PDF, – сгенерированные с помощью системы LaTeX. Опуская многочисленные детали (в частности, различные флаги `latex2html`), приведу упрощенный makefile, предназначенный для выполнения этой процедуры.



Если вы копируете эти фрагментарные кусочки с экрана или с бумаги в файл makefile, имейте в виду, что красная строка формируется с помощью знака табуляции, а не пробелов. Так решил POSIX.

```
all: html doc publish

doc:
pdflatex $(f).tex

html:
latex -interaction batchmode $(f)
latex2html $(f).tex

publish:
scp $(f).pdf $(Blogserver)
```

Значение переменной f я задавал из командной строки, например export f=tip-make. При запуске make без параметров проверяется первая по порядку цель, all. Иными словами, просто make эквивалентно make first_target. Цель all зависит от целей html, doc и publish, они и проверяются именно в таком порядке. Если я еще не готов отправить свое творение в мир, то могу набрать команду make html doc, которая выполнит лишь указанные шаги.

В простом показанном выше makefile есть только одна группа цель/зависимость/скрипт. Например:

```
P=domath
OBJECTS=addition.o subtraction.o

$(P): $(OBJECTS)
```

Это похоже на последовательность зависимостей и скриптов в makefile для публикации статьи в блоге, только скрипты неявные. В данном случае P=domath – подлежащая компиляции программа, она зависит от объектных файлов *addition.o* и *subtraction.o*. Поскольку *addition.o* не встречается в виде цели, make применяет неявное правило: компилировать *c*-файл в *o*-файл. То же самое делается для *subtraction.o* и *domath.o* (потому что GNU make неявно предполагает, что в представленной ситуации domath зависит от *domath.o*). После построения всех объектных файлов выясняется, что скрипт для построения цели $(P) не указан, поэтому GNU make подставляет свой скрипт по умолчанию, который компонует из *o*-файлов исполняемую программу.

У совместимой с POSIX программы make имеется специальный рецепт для компиляции объектного *o*-файла из исходного *c*-файла:

```
$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $*.c
```

Здесь переменная $(CC) представляет компилятор C; в стандарте POSIX определено, что по умолчанию CC=c99, но в современных версиях GNU make считается, что CC=cc, а cc обычно является ссылкой на gcc. В минимальном файле makefile в начале этого раздела переменной $(CC) явно присвоено значение c99, переменной $(CFLAGS) – набор флагов, а переменная $(LDFLAGS) вообще не задана, поэтому вместо нее подставляется пустая строка. Таким образом, если make определит, что нужно построить файл your_program.o, то при таком makefile будет выполнена следующая команда:

```
c99 -g -Wall -O3 -o your_program.o your_program.c
```

Если GNU make решит, что требуется построить исполняемую программу из объектных файлов, то она воспользуется следующим встроенным рецептом:

```
$(CC) $(LDFLAGS) first.o second.o $(LDLIBS)
```

Напомним, что порядок файлов важен для компоновщика, поэтому нам нужны две относящиеся к нему переменные, одной не обойтись. В нашем примере для компоновки нужно было бы выполнить команду

```
cc specific.o -lbroad -lgeneral
```

Сравнив ее с рецептом, мы видим, что нужно следующим образом задать переменную LDLIBS=-lbroad -lgeneral. Если бы мы задали LDFLAGS=-lbroad -lgeneral, то после подстановки переменной в рецепт получилась бы команда cc -lbroad -lgeneral specific.o, которая, скорее всего, не приведет к желаемому результату. Отметим, что LDFLAGS входит также в рецепт компиляции *c*-файлов в *o*-файлы.



Чтобы получить полный перечень встроенных в вашу версию make правил и переменных, выполните команду

```
make -p > default_rules
```

Итак, игра состоит в том, чтобы определить, какие переменные нужны, и задать их в makefile. Вам все же придется провести кое-какие изыскания, чтобы узнать правильные значения флагов, но, по крайней мере, их можно будет один раз записать в makefile и больше об этом не думать.



Ваша очередь. Модифицируйте makefile для компиляции *erf.c*.

При работе с IDE, CMAKE или иными альтернативами совместимой с POSIX программе make в игру «найди переменные» играть все равно придется. Я продолжу обсуждение минимального makefile, и у вас не должно возникнуть сложностей с определением соответствующих переменных в своей IDE.

- Переменная CFLAGS давно и прочно укоренилась в практике, но переменная, определяющая флаги компоновщика, меняется от системы к системе. Даже переменная LDLIBS не определена в POSIX, но именно она используется в GNU make.
- В переменных CFLAGS и LDLIBS мы будем задавать все флаги компилятора и компонуемые библиотеки. Если в вашей системе есть программа pkg-config, воспользуйтесь обратными апострофами. Например, в своей системе я почти все программы компоную с библиотеками Apophenia и GLib, поэтому makefile у меня выглядит так:

  ```
  CFLAGS=`pkg-config --cflags apophenia glib-2.0` -g -Wall -std=gnu11 -O3
  LDLIBS=`pkg-config --libs apophenia glib-2.0`
  ```

  Или можно задать флаги -I, -L и -l вручную:

  ```
  CFLAGS=-I/home/b/root/include -g -Wall -O3
  LDLIBS=-L/home/b/root/lib -lweirdlib
  ```

- Если вы добавили местоположение библиотеки и соответствующих ей заголовков в строки LDLIBS и CFLAGS и знаете, что в вашей системе это работает, то не имеет смысла впоследствии удалять эту информацию. Так ли уж важно, что конечный исполняемый файл будет на 10 килобайт длиннее, чем мог бы получиться при создании специального makefile для каждой новой программы? Это означает, что можно написать один makefile, содержащий сведения обо всех библиотеках в системе, и копировать его из проекта в проект без изменений.

- Если имеется два (или более) С-файла, добавьте `second.o third.o` и т. д. в строку `OBJECTS` (запятые не нужны, имена разделяются только пробелами) в начале показанного выше файла makefile. Программа `make` использует эту информацию, чтобы понять, какие файлы строить и с помощью каких рецептов.
- Если программа состоит всего из одного *c*-файла, то makefile вообще необязателен. Находясь в каталоге, где есть только файл *erf.c*, но нет makefile, выполните следующие команды в оболочке:

```
export CFLAGS='-g -Wall -O3 -std=gnu11'
export LDLIBS='-lm'
make erf
```

и полюбуйтесь, как `make` пользуется своими знаниями о компиляции С, чтобы сделать все остальное.

Какие флаги компоновщика задавать при построении разделяемой библиотеки?

Честно говоря, не знаю. В разных операционных системах (отличающихся как по типу, так и по году выпуска) и даже в одной системе правила зачастую различаются, и разобраться в них нелегко.
Лучше пользуйтесь инструментом *Libtool*, описанным наряду с прочими в главе 3, уж он-то знает все о генерации разделяемых библиотек в любой операционной системе. Я рекомендую потратить время на изучение Autotools и тем решить проблему компиляции разделяемых библиотек раз и навсегда. Это полезнее, чем тратить время на знакомство с флагами компилятора и процедурой компоновки для каждой целевой системы.

Сборка библиотек из исходного кода

До сих пор мы говорили о компиляции своей программы с помощью `make`. Компиляция же чужого кода – совершенно другая история.

Попробуем на примере какого-нибудь пакета. Библиотека GNU Scientific Library (GSL) включает великое множество функций для численных расчетов.

Пакет GSL сформирован с помощью *Autotools*, комплекта инструментов для подготовки библиотеки к использованию на любой машине. Своей цели этот комплект достигает путем проверки всех известных платформенных причуд и выбора подходящего обходного решения. В наши дни подавляющее большинство программ распространяется с помощью Autotools; подробно о том, как подготовить пакет для своей программы вместе с библиотеками, будет рассказано ниже в разделе «Подготовка пакета с помощью Autotools». А пока будем выступать в роли пользователей и порадуемся тому, как легко и быстро можно установить полезные библиотеки.

GSL часто можно получить в готовом виде с помощью менеджера пакетов, но в педагогических целях покажем, как скачать исходный код GSL и откомпилировать его в предположении, что у вас есть привилегии суперпользователя root на своем компьютере.

```
wget ftp://ftp.gnu.org/gnu/gsl/gsl-1.15.tar.gz ❶
tar xvzf gsl-*gz                               ❷
cd gsl-1.15
./configure                                    ❸
make
sudo make install                              ❹
```

❶ Скачать сжатый архив. С помощью менеджера пакетов установить программу wget, если ее нет в системе, или перейти по указанному URL-адресу в браузере.
❷ Распаковать архив: x=извлечь, v=подробная информация, z=распаковывать с помощью gzip, f=имя файла.
❸ Определить причуды машины. Если configure выводит сообщение об ошибке, вызванной отсутствием какого-то элемента, установить его с помощью менеджера пакетов и запустить configure снова.
❹ Установить в нужное место – при наличии прав.

Если вы делаете все это дома, то, наверное, привилегии root у вас есть, и все отработает на ура. Если же вы упражняетесь на работе, используя разделяемый сервер, то маловероятно, что вам дали права суперпользователя, поэтому вы не сможете ввести пароль, который запрашивается на последнем шаге. В таком случае потерпите до следующего раздела.

Установили? В примере 1.3 приведена короткая программа, которая пытается найти 95-процентный доверительный интервал с помощью функции из библиотеки GSL; попробуйте собрать ее и запустить.

Пример 1.3 ❖ Переработка примера 1.1 с применением GSL (gsl_erf.c)

```
#include <gsl/gsl_cdf.h>
#include <stdio.h>

int main(){
    double bottom_tail = gsl_cdf_gaussian_P(-1.96, 1);
    printf("Area between [-1.96, 1.96]:%g\n", 1-2*bottom_tail);
}
```

Чтобы воспользоваться только что установленной библиотекой, необходимо изменить makefile для сборки программы, в которой эта библиотека используется, указав пути к файлам.

В зависимости от того, есть в системе pkg-config или нет, нужно сделать одно из двух:

```
LDLIBS=`pkg-config --libs gsl`
# или
LDLIBS=-lgsl -lgslcblas -lm
```

Если библиотека установилась не в стандартное место и pkg-config отсутствует, то нужно будет добавить пути в переменные, задаваемые в начале файла, например: CFLAGS=-I/usr/local/include и LDLIBS=-L/usr/local/lib -Wl,-R/usr/local/lib.

Сборка библиотек из исходного кода (даже если системный администратор против)

Вы, конечно, обратили внимание на сделанную выше оговорку о необходимости иметь привилегии root для установки в обычные места, определенные в стандарте POSIX. Но при работе с разделяемым компьютером в офисе или по какой-то иной причине таких привилегий у вас может и не быть.

Тогда придется уходить в подполье и создавать свой личный корневой каталог. Первым делом нужно просто создать этот каталог:

```
mkdir ~/root
```

У меня уже есть каталог *~/tech*, в котором я храню разного рода технические средства, руководства и фрагменты кода, поэтому я создал в нем подкаталог *~/tech/root*. Имя не имеет значения, мне просто понравилось имя *~/root*.



Оболочка подставляет вместо тильды полный путь к домашнему каталогу, избавляя от необходимости набирать лишнее. Стандарт POSIX требует лишь, чтобы подстановка производилась только в начале слова или после двоеточия (последнее необходимо для перечисления путей в переменной `PATH`), но многие оболочки делают это и тогда, когда тильда находится в середине слова. Другие программы, в частности `make`, могут как распознавать, так и не распознавать тильду.

Следующий шаг – добавить всюду, куда нужно, путь к каталогу в нашей новой «корневой файловой системе». Что касается программ, то это переменная `PATH` в файле *.bashrc* (или эквивалентном ему):

```
PATH=~/root/bin:$PATH
```

Поместив путь к подкаталогу *bin* в новом корневом каталоге в начало исходной `PATH`, мы гарантируем, что при поиске программ сначала будет просматриваться именно он. Поэтому вы можете заместить любую программу, которая уже находится в стандартном системном каталоге, другой, предпочтительной для вас, версией.

Чтобы ваши С-программы компоновались с нужными вам библиотеками, пополните пути поиска в рассмотренном выше makefile:

```
LDLIBS=-L/home/your_home/root/lib (затем прочие флаги: -lgsl -lm ...)
CFLAGS=-I/home/your_home/root/include (плюс -g -Wall -O3 ...)
```

Сформировав свой локальный корень, вы можете использовать его и в других системах, например в переменной `CLASSPATH`, применяемой в Java.

Последний шаг – установка программ в новый корень. Если у вас имеется исходный код, подготовленный с помощью Autotools, то нужно лишь добавить флаг `--prefix=$HOME/root`:

```
./configure --prefix=$HOME/root; make; make install
```

На шаге установки `sudo` не потребуется, потому что вы находитесь на собственной территории.

Поскольку программы и библиотеки «живут» в вашем домашнем каталоге и имеют не больше прав, чем вы сами, системный администратор не вправе ругаться,

что они могут кому-то навредить. Если сисадмин все-таки недоволен, то, как это ни грустно, придется с ним поссориться.

Руководство

Допускаю, что были времена, когда руководство представляло собой бумажный документ, но в наши дни оно существует в виде команды `man`. Например, чтобы прочитать о функции `strtok`, нужно набрать `man strtok`, и вы узнаете, какой заголовок включать в программу, какие аргументы функция принимает и как она должна использоваться. Страницы руководства написаны конспективно, иногда в них нет примеров и вообще предполагается, что читатель уже имеет общее представление о том, что делает функция. Если нужно более подробное пособие, то поисковик, скорее всего, предложит несколько ресурсов в Интернете (а в случае конкретно `strtok` смотрите раздел «Песнь о strtok» ниже. Руководство по библиотеке GNU C, которое также легко найти в Сети, написано очень понятно и рассчитано на начинающих.

- Если не можете вспомнить, как называется то, что вы ищете, то можно воспользоваться однострочным рефератом, присутствующим в каждой странице руководства. Команда `man -k searchterm` производит поиск по этим рефератам. Во многих системах есть также команда `apropos`, аналогичная `man -k`, но с некоторыми дополнительными возможностями. Для выделения того, что меня интересует, я обычно подаю выход `apropos` на вход `grep`.
- Руководство разбито на разделы. В разделе 1 собраны программы, запускаемые из командной строки, а в разделе 3 – библиотечные функции. Если в системе имеется программа `printf`, то команда `man printf` покажет документацию по ней, а команда `man 3 printf` – документацию по функции `printf` из стандартной библиотеки C.
- Для получения дополнительных сведений о команде `man` (в том числе полного списка разделов) наберите `man man`.
- В ваш текстовый редактор или IDE могут быть встроены средства быстрого поиска страниц руководства. Например, при работе с редактором vi можно подвести курсор к слову и нажать *K* – откроется страница руководства по этому слову.

Компиляция C-программы с помощью встроенного документа

Мы уже несколько раз видели общую последовательность компиляции.

1. Задать переменную, содержащую флаги компилятора.
2. Задать переменную, содержащую флаги компоновщика, в том числе по одному флагу `-l` для каждой используемой библиотеки.
3. С помощью рецептов, встроенных в `make` или IDE, преобразовать переменные в полные команды компиляции и компоновки.

В оставшейся части главы мы проделаем все это еще один, последний раз, сведя конфигурацию к абсолютному минимуму: с помощью одной лишь оболочки. Если вы из тех людей, что учат скриптовые языки путем копирования фрагментов кода в интерпретатор, то сможете точно так же скопировать написанный на C код в командную строку.

Включение файлов-заголовков из командной строки

В gcc и в Clang есть удобный флаг для включения заголовков. Например,

```
gcc -include stdio.h
```

эквивалентно строке

```
#include <stdio.h>
```

в начале C-файла; аналогично `clang -include stdio.h`.

Добавив сюда вызов компилятора, мы наконец сможем записать программу *hello.c* в виде одной строки, каковой она и должна быть:

```
int main(){ printf("Hello, world.\n"); }
```

а откомпилировать так:

```
gcc -include stdio.h hello.c -o hi --std=gnu99 -Wall -g -O3
```

или воспользовавшись командами оболочки:

```
export CFLAGS='-g -Wall -include stdio.h'
export CC=c99
make hello
```

Такое использование флага `-include` зависит от компилятора и подразумевает перенос части информации из кода в инструкции компиляции. Если вы считаете подобную практику порочной, забудьте про этот совет.

Универсальный заголовок

Я хотел бы отвлечься от основной темы и посвятить несколько абзацев файлам-заголовкам. Полезный заголовок должен включать псевдонимы типов, определения макросов и объявления функций, которые необходимы коду, пользующемуся этим заголовком. С другой стороны, он не должен включать псевдонимов типов, определений макросов и объявлений функций, которых код использовать не будет.

Чтобы в точности удовлетворить оба этих условия, придется писать отдельный заголовок для каждого файла с кодом, включая те и только те части, которые в этом коде используются. Но так никто не делает. В каких-то случаях заголовок – это набор частей, которые, по мнению автора, связаны между собой. А иногда заголовок содержит весь открытый интерфейс некоторой библиотеки. Возможно, в вашем проекте есть какие-то части, востребованные в разных местах, тогда вы заводите для них внутренний заголовок.

Было время, когда компилятору требовалось несколько секунд или минут, чтобы откомпилировать даже сравнительно простую программу, поэтому любая попытка сократить объем выполняемой компилятором работы имела заметный человеку эффект. Но теперь файлы *stdio.h* и *stdlib.h* на моей машине содержат примерно 1000 строк каждый (наберите `wc -l /usr/include/stdlib.h`), а файл *time.h* – еще 400 строк, то есть такая вот программа из семи строчек:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    srand(time(NULL));          // Инициализировать генератор случайных чисел.
    printf("%i\n", rand());    // Напечать число.
}
```

на самом деле занимает ~2400 строк.

Но компилятор уже не считает, что 2400 строк – большая проблема, на компиляцию этой программы уходит меньше секунды. Так зачем тратить время на скрупулезный отбор строго необходимых программе заголовков?

Ниже будут приведены примеры, в которых используется библиотека Glib, в них в начале стоит директива `#include <glib.h>`. Этот заголовок включает 74 подзаголовка, охватывающих все разделы Glib. И это правильный подход, потому что те из нас, кто не хочет терять времени на выбор только нужных разделов библиотеки, могут ограничиться одной строкой, включающей все, а те, кому необходим детальный контроль, могут включить именно те заголовки, которые реально необходимы. Хорошо было бы иметь такой универсальный заголовок и для стандартной библиотеки C; в начале 1980-х годов это было не принято, но легко изготовить его самостоятельно.



Ваша очередь. Напишите единственный заголовок, назвав его, скажем, *allheads.h*, в который включите все заголовки, которые используете. Например:

```
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <gsl/gsl_rng.h>
```

Не могу точно сказать, как он должен выглядеть, потому что не знаю, какими заголовками вы пользуетесь в своей работе.

Создав агрегированный заголовок, вы можете помещать такую строку

```
#include <allheads.h>
```

в начало любого программного файла и больше вообще не думать о заголовках. Конечно, при его раскрытии получится, быть может, 10 000 строк лишнего кода, не имеющего никакого отношения к вашей программе. Но вы этого не заметите, потому что неиспользуемые объявления не отражаются на конечном исполняемом файле.

Если вы пишете заголовок, предназначенный для других людей, то в соответствии с правилом отсутствия ненужных элементов он не должен содержать директивы `#include "allheads.h"`, подгружающей все определения и объявления из стандартной библиотеки. Более того, ваш открытый заголовок может вообще не содержать ни одного элемента из стандартной библиотеки. Это общее правило: в вашей библиотеке может быть код, в котором используются связанные списки из Glib, но тогда директива `#include <glib.h>` должна находиться в файле кода, а не в открытом заголовке библиотеки.

Возвращаясь к идее быстрой компиляции из командной строки, отмечу, что универсальный заголовок ускоряет написание простеньких программ. При наличии такого заголовка в среде gcc или Clang становится лишней даже строка `#include <allheads.h>`, потому что можно вместо нее добавить флаг `-include allheads.h` в переменную `CFLAGS` и не думать о том, какие не относящиеся к самому проекту заголовки включать.

Встроенные документы

Встроенные документы поддерживаются во всех POSIX-совместимых оболочках, их можно использовать при программировании на C, Python, Perl и любом другом

языке. В этой книге они встречаются часто, внося в текст некоторое оживление. Кроме того, встроенные документы облегчают создание многоязыковых скриптов. Никто не мешает произвести синтаксический разбор на Perl, выполнить математические вычисления на C, а затем использовать Gnuplot для создания красивых картинок – и все это в одном текстовом файле.

Вот пример на Python. Обычно написанный на этом языке скрипт запускается следующим образом:

```
python your_script.py
```

Python позволяет задать специальное имя файла -, обозначающее стандартный ввод:

```
echo "print 'hi.'" | python -
```

Теоретически можно было бы с помощью команды echo разместить в командной строке весьма длинные скрипты, но вы быстро устанете от мелких, но нежелательных отвлечений; например, нужно писать \"hi\" вместо "hi".

На выручку приходит *встроенный документ*, в котором ничего экранировать не нужно. Вот пример:

```
python - <<"XXXX"
lines=2
print "\nThis script is%i lines long.\n"%(lines,)
XXXX
```

- Встроенные документы – стандартная возможность оболочки, поэтому должны работать в любой POSIX-совместимой системе.
- Вместо "XXXX" можно подставить любую другую комбинацию символов, например очень популярна "EOF", а "-----" отлично смотрится при условии, что количество дефисов в начале и в конце совпадает. Когда оболочка видит строку, состоящую только из выбранной комбинации символов, она прекращает подавать скрипт на стандартный ввод программы. И никакого другого анализа не производит.
- Существует также вариант, начинающийся с <<-. В этом случае удаляются все знаки табуляции в начале каждой строки, поэтому встроенный документ можно поместить в секцию скрипта оболочки, имеющую отступ, не нарушая структуры отступов. Разумеется, для встроенного документа, содержащего код на Python, это было бы катастрофой.
- Отметим также различие между <<"XXXX" и <<XXXX. Во втором случае оболочка разбирает некоторые элементы, то есть может подставить значения переменных вида $shell_variable. В оболочке символ $ активно используется для обозначения переменных и других конструкций, но в языке C этот символ не имеет специального значения. Похоже, люди, которые писали Unix, спроектировали ее сразу от начала до конца, чтобы легко было писать скрипты оболочки, порождающие код на C...

Компиляция из stdin

Но вернемся к C: мы можем использовать встроенные документы для компиляции с помощью gcc или Clang C-кода, вставленного в командную строку. А можем включить несколько строк на C в многоязычный скрипт.

Мы не собираемся использовать makefile, поэтому понадобится однострочная команда компиляции. Чтобы упростить себе жизнь, заведем для нее псевдоним. Выполните следующие команды вручную или поместите их в свой файл *.bashrc*, *.zshrc* или подходящий эквивалент:

```
go_libs="-lm"
go_flags="-g -Wall -include allheads.h -O3"
alias go_c="c99 -xc - $go_libs $go_flags"
```

Здесь *allheads.h* – созданный ранее агрегированный заголовок. Флаг `-include` позволяет не думать о заголовках при написании кода на C, и, как я обнаружил, история bash сбоит, если в C-коде встречается символ `#`.

В строке компиляции вы видите уже знакомый дефис `-`, означающий, что текст читается не из файла, а из stdin. Флаг -xc говорит, что этот текст следует интерпретировать как код на C. Поскольку *gcc* означает «GNU Compiler Collection» (набор компиляторов GNU), а не «GNU C Compiler» (компилятор GNU C), то без этого флага невозможно понять, что речь идет о коде на C, так как нет имени файла, оканчивающегося на `.c`. Следовательно, мы должны явно сообщить, что это код не на Java, Fortran, Objective C, Ada или C++ (то же справедливо и в отношении Clang, пусть даже это название вызывает ассоциации с *C language*).

Все, что вы делали при настройке переменных `LDLIBS` и `CFLAGS` в makefile, повторите здесь.

Теперь все готово и можно откомпилировать C-код прямо в командной строке:

```
go_c << '---'
int main(){printf("Hello from the command line.\n");}
---
./a.out
```

Мы можем воспользоваться встроенным документом для вставки коротких C-программ в командную строку, что позволяет без лишних хлопот писать небольшие тестовые программы. Не нужен не только makefile, но даже и входной файл[1].

Не думайте, что все вышеописанное должно стать основным способом работы. Но вставлять фрагменты кода в командную строку забавно, а возможность вставлять коротенький код на C в длинный скрипт оболочки – это вообще чудо.

---

[1] В POSIX определено соглашение о том, что если первая строка файла имеет вид

```
#!aninterpreter
```

то при запуске такого файла из оболочки исполняется программа aninterpreter. Это очень удобно для интерпретируемых языков типа Perl или Python (особенно если учесть, что знак # означает в них начало комментария, так что первая строка игнорируется). Пользуясь советами из этого раздела, вы могли бы написать скрипт, скажем c99sh, который делал бы то, что требуется, для C-файла, начинающегося строкой #!c99sh: отрезал бы первую строку, все остальное передавал бы по конвейеру компилятору, а в конце запускал бы сгенерированную программу. Впрочем, Рис Улерих уже написал такой скрипт и опубликовал его в Github.

Глава 2

Отладка, тестирование, документирование

Ползу
По твоему окну
Ты мнишь, что мне неловко,
А я вот жду...
Чтоб завершить свою уловку.
— Wire «I Am the Fly»

В этой главе мы рассмотрим средства отладки, тестирования и документирования вашего творения – то, что необходимо для доведения потенциально полезного собрания скриптов до чего-то, на что вы сами и другие люди смогут положиться.

Поскольку C оставляет программисту свободу творить в памяти совершенно немыслимые вещи, отладка является одновременно банальной проверкой логики (с помощью gdb) и технически более сложной задачей вылавливания неправильного выделения и утечек памяти (с помощью Valgrind). В плане документирования мы рассмотрим один инструмент на уровне интерфейса (Doxygen) и другой, который помогает документировать и разрабатывать каждый шаг программы (CWEB).

В этой главе мы также вкратце коснемся *тестовой оснастки* – комплекта средств, позволяющего быстро писать многочисленные тесты программы. И завершим главу рассмотрением стратегии уведомления об ошибках и принципов обработки данных, вводимых пользователем, и ошибок в них.

Работа с отладчиком

Первый совет касательно отладчика будет прост и краток:

Пользуйтесь отладчиком, обязательно.

Кто-то скажет, что это и не совет вовсе, потому что кто же не пользуется отладчиком? Но поскольку это второе издание книги, могу сообщить, что одной из самых часто повторяющихся просьб было включить более подробное введение в работу с отладчиком – для многих читателей это было откровением.

Некоторые считают, что обычно ошибки – результат неправильного понимания в широком смысле, тогда как отладчик дает только низкоуровневую информацию о состоянии переменных и стека вызовов. Действительно, отыскав место ошибки в отладчике, необходимо остановиться и подумать о том, что привело к этой

ошибке и не проявится ли та же проблема в каком-нибудь другом месте. Иногда в свидетельство о смерти включают глубокий анализ причины: *смерть произошла в результате __________, что явилось результатом ________, что явилось результатом ________, что явилось результатом ________*. После того как с помощью отладчика вы провели такой анализ и стали лучше понимать свою программу, обретенное знание следует инкапсулировать в дополнительных автономных тестах.

Теперь что касается слова «обязательно». Прогон программы под отладчиком обходится практически бесплатно. И не стоит извлекать отладчик на свет божий, только когда что-то ломается. Линус Торвальдс говорит: «Я пользуюсь отладчиком постоянно... вроде как накачанным дизассемблером, который еще и программировать можно»[1]. Ну неужели вас не прельщает возможность остановиться в любом месте, повысить уровень детализации вывода простой командой `print verbose++`, досрочно выйти из цикла `for (int i=0; i<10; i++)`, введя команды `print i=100` и `continue`, или протестировать функцию, подав ей на вход различные значения? Любители интерактивных языков правы в том, что взаимодействие с кодом улучшает процесс разработки во всех отношениях; но они так и не удосужились дочитать учебник C до главы об отладке и не знают, что все эти интерактивные штучки применимы и к C.

Для чего бы вы ни использовали отладчик, он должен уметь представлять хранящуюся в программе отладочную информацию (например, имена переменных и функций) в понятном человеку виде. Чтобы включить в исполняемый файл отладочные символы, при компиляции следует указать флаг `-g` (например, в переменной `CFLAGS`). Причин не использовать флаг `-g` очень мало – он не замедляет работу программы, а увеличение размера файла на килобайт-другой в большинстве случаев несущественно. Отладка также упрощается при отключении оптимизации с помощью флага `-O0` (O нуль), потому что иногда оптимизатор устраняет переменные, которые могли бы оказаться полезны при отладке, да и вообще видоизменяет код всякими неожиданными способами.

Я в основном рассматриваю GDB, потому что в большинстве POSIX-совместимых систем ничего другого просто нет[2]. Отладчик LLDB (поставляемый вместе с LLVM/Clang) постепенно набирает популярность, и я расскажу о нем тоже. Компания Apple перестала включать GDB в свою среду Xcode, но его можно установить с помощью менеджера пакетов, например Macports, Fink или Homebrew. В Mac сеансы отладки, возможно, придется запускать через sudo(!), например `sudo lldb stddev_bugged`.

Быть может, вы работаете в IDE или другой графической среде, которая запускает вашу программу под отладчиком всякий раз, как вы выбираете из меню команду «Выполнить». Я буду демонстрировать только работу из командной строки, но

[1] Из письма Торвальдса к коллеге от 6 сентября 2000 года.

[2] Кстати, компилятор C++ подправляет (mangle) имена функций. В GDB это видно, и я всегда считал отладку кода на C++ в GDB мучительным делом. Но компилятор C ничего подобного не делает, поэтому для него с GDB работать куда проще, и не нужны вспомогательные средства для восстановления имени в исходном виде.

вряд ли вас затруднит перевод команд на язык щелчков мышью. Некоторые графические фасады позволяют использовать макросы, определенные в файле *.gdbinit*.

При работе непосредственно с командной строкой вам, возможно, будет удобно видеть код в текстовом редакторе в соседнем открытом окне или на другом терминале. Простая комбинация отладчика с редактором дает многие преимущества IDE и, может статься, больше вам ничего и не понадобится.

Стек кадров

Для запуска программы необходимо попросить систему выполнить функцию main. Компьютер генерирует *кадр*, в котором хранится информация о вызове функции, в том числе ее входные параметры (которые в случае main принято называть argc и argv) и созданные внутри нее локальные переменные.

Допустим, что в процессе выполнения main вызывает функцию get_agents. В этот момент выполнение main приостанавливается и генерируется новый кадр для get_agents, где хранятся детали ее вызова. Быть может, get_agents, в свою очередь, вызывает функцию agent_address, и таким образом мы получаем растущий *стек* кадров. Рано или поздно выполнение agent_address завершится, в этот момент ее кадр будет вытолкнут из стека, и возобновится выполнение get_agents.

На вопрос «Где я нахожусь?» проще всего ответить, указав номер строки в программе, и иногда этого достаточно. Однако чаще вас интересует «Как я сюда попал?», и ответом на этот вопрос является *обратная трассировка*, или *стек вызовов*, то есть стек кадров. Вот пример обратной трассировки:

```
#0 0x00000000000413bbe in agent_address (agent_number=312) at addresses.c:100
#1 0x00000000004148b6 in get_agents () at addresses.c:163
#2 0x0000000000404f9b in main (argc=1, argv=0x7fffffffe278) at addresses.c:227
```

На вершине стека находится кадр 0, а на самом дне – вызов main, который в данный момент находится в кадре 2 (но номер кадра будет меняться по мере роста и сокращения стека). Шестнадцатеричное число после номера кадра – это адрес, с которого возобновится выполнение после возврата из вызванной функции; для меня как прикладного программиста это только зрительный шум, я на него не обращаю внимания. Далее показаны имя функции, ее входные параметры (в случае argv это опять-таки шестнадцатеричный адрес) и номер строки в исходном коде.

Если вы обнаружили, что номер дома в адресе агента (agent_address) заведомо неправилен, то, быть может, это потому, что передан неправильный номер агента (agent_number), и в этом случае стоит перейти в кадр 1 и поинтересоваться, в каком состоянии находилась get_agents и почему получилось такое странное состояние agent_address. В значительной мере искусство исследования программы заключается в перемещении по стеку и прослеживании причин и следствий между кадрами.

Отладка программы как детективная история

В этом разделе мы рассмотрим воображаемый сеанс вопросов и ответов с применением GDB или LLDB. В примерах к этой книге имеется файл stddev_bugged.c, вариант в примере 7.4 с ошибкой. Как в любом хорошем детективе, все ключи, необходимые для изобличения преступника, перед вами. Правильно выстроенная последовательность вопросов поможет исключить подозреваемых одного за другим, пока не останется всего один и ошибка не станет очевидной.

После компиляции программы (с помощью команды CFLAGS="-g" make stddev_bugged) приступим к расследованию и для начала запустим отладчик:

```
gdb stddev_bugged
# или
lldb stddev_bugged
```

Перед нами приглашение к вводу команд отладчика, можно задавать вопросы.

В. Что делает эта программа?

О. Команда run запускает программу. Ее, как и все команды GDB и LLDB, можно записать в сокращенном виде:

```
(gdb) r
mean: 5687.496667 var: 194085710
mean: 0.83 var: 4.1334
[Inferior 1 (process 22734) exited normally]
```

Похоже, программа вычисляет какие-то средние значения и дисперсии. Она дошла до конца без нарушения защиты памяти и прочих ошибок и вернула нуль, что означает нормальное завершение.

В. Проверяет ли код в main правильность результата?

О. Посмотреть на исходный код проще всего, открыв его в текстовом редакторе. Существуют способы расположить редактор рядом с отладчиком даже при удаленной работе на машине, оснащенной только текстовыми терминалами (см. врезку «Визуальные средства» на стр. 116). Но GDB и LLDB тоже умеют показывать фрагменты кода, для этого служит команда list:

```
(gdb) l main

28        }
29        return (meanvar){.mean = avg,
30                         .var = avg2 - pow(avg, 2)}; //E[x^2] - E^2[x]
31     }
32
33     int main(){
34        double d[] = { 34124.75, 34124.48,
35                       34124.90, 34125.31,
36                       34125.05, 34124.98, NAN};
37
```

Выведено десять строк программы выше и ниже запрошенной. При повторном выполнении команды list без аргументов будут выведены следующие 10 строк:

```
(gdb) l
38        meanvar mv = mean_and_var(d);
39        printf("mean:%.10g var:%.10g\n", mv.mean, mv.var*6/5.);
40
41        double d2[] = { 4.75, 4.48,
42                        4.90, 5.31,
43                        5.05, 4.98, NAN};
44
45        mv = mean_and_var(d2);
46        mv.var *= 6./5;
47        printf("mean:%.10g var:%.10g\n", mv.mean, mv.var);
```

В строке 38 мы видим обращение к функции mean_and_var, которой передается список d. И тут, очевидно, имеет место проблема: все числа в d расположены в окрестности 34 125, а напечатанное программой среднее значение приближенно равно 5687, (не говоря уже о какой-то несуразной дисперсии). Аналогично при втором обращении функции mean_and_var передается список чисел в окрестности 5, а среднее оказалось равно 0.83.

Вся оставшаяся часть сеанса посвящена поиску ответа на единственный вопрос: *где то первое место в программе, начиная с которого все пошло наперекосяк*? Но чтобы ответить на этот главный вопрос, нам нужны дополнительные детали.

В. Как узнать, что происходит внутри mean_and_var?

О. Мы хотим приостановить программу при входе в mean_and_var, чтобы поставить там точку останова:

```
(gdb) b mean_and_var
Breakpoint 1 at 0x400820: file stddev_bugged.c, line 16.
```

Поставив точку останова, заново запустим программу – она остановится в этой точке:

```
(gdb) r
Breakpoint 1, mean_and_var (data=data@entry=0x7fffffffe130) at
stddev_bugged.c:16
16 meanvar mean_and_var(const double *data){
(gdb)
```

Сейчас мы стоим в строке 16, в самом начале функции, и можем задавать дальнейшие вопросы о том, что в ней происходит.

В. В переменной data находится то, что мы думаем?

О. Посмотреть на переменную data в текущем кадре позволяет команда print, сокращенно p:

```
(gdb) p *data
$2 = 34124.75
```

Печально – нам показали только первый элемент. Однако в GDB имеется специальная конструкция @- для печати последовательности элементов массива. Вот как запросить первые 10 элементов [LLDB: mem read -tdouble -c10 data]:

```
(gdb) p *data@10
$3 = {34124.75,
  34124.480000000003,
  34124.900000000001,
  34125.309999999998,
  34125.050000000003,
  34124.980000000003,
  nan(0x8000000000000),
  7.7074240751234461e-322,
  4.9406564584124654e-324,
  2.0734299798669383e-317}
```

Обратите внимание на звездочку в начале выражения, без нее мы получили бы последовательность из десяти шестнадцатеричных адресов.

Я запросил 10 элементов, потому что было лень считать, сколько элементов хранится в наборе данных, но первые семь из десяти выглядят правильно: последовательность чисел, в конце которой находится маркер NaN. После него мы видим мусор – неинициализированную память за концом массива.

В. Соответствует ли это тому, что мы передали из `main`?

О. Команда `bt` печатает обратную трассировку:

```
(gdb) bt
#0 mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugged.c:16
#1 0x0000000000400680 in main () at stddev_bugged.c:38
```

В стеке находятся всего два кадра: текущий и вызвавший его, `main`. Посмотрим на данные в кадре 1 и для начала переключимся на него:

```
(gdb) f 1
#1 0x0000000000400680 in main () at stddev_bugged.c:38
38 meanvar mv = mean_and_var(d);
```

Сейчас отладчик находится в кадре функции `main`, в строке 38. Это ожидаемое место, так что порядок выполнения правильный (и не изменен оптимизатором). Находясь в этом кадре, посмотрим на массив данных с именем `d`:

```
(gdb) p *d@7
$5 = {34124.75,
  34124.480000000003,
  34124.900000000001,
  34125.309999999998,
  34125.050000000003,
  34124.980000000003,
  nan(0x8000000000000)}
```

Данные совпадают с теми, что мы видели в кадре `mean_and_var`, так что с набором данных вроде бы ничего странного не произошло.

Нам нет необходимости явно возвращаться в кадр 0, чтобы продолжить выполнение программы, но это можно было бы сделать командой `f 0` или командой перемещения по стеку относительно текущего кадра:

```
(gdb) down
```

Отметим, что в командах `up` и `down` предполагается числовой порядок. Если считать, что в списке, который выводит `bt` (как в GDB, так и в LLDB), кадр с наименьшим номером располагается сверху, то `up` идет вниз, а `down` вверх по списку обратной трассировки.

В. Эта проблема случайно не связана с параллельными потоками?

О. Получить список потоков позволяет команда `info threads` [LLDB: `thread list`]:

```
(gdb) info threads
Id   Target Id          Frame
```

```
* 1  Thread 0x7ffff7fcb7c0 (LWP 28903) "stddev_bugged" mean_and_var
(data=data@entry=0x7fffffffe180) at stddev_bugged.c:16
```

В данном случае существует всего один активный поток, поэтому проблема никак не может быть связана с многопоточностью. Символ * показывает, в каком потоке сейчас находится отладчик. Если бы существовал поток 2, то мы могли бы перейти в него командой `thread 2` (GDB) или `thread select 2` (LLDB).



Если до сих пор вы в своих программах не запускали несколько потоков, то после прочтения главы 12 ситуация обязательно изменится. Пользователям GDB рекомендуется добавить в файл `.gdbinit` показанную ниже команду, чтобы отключить надоедливые уведомления о каждом создании нового потока:

```
set print thread-events off
```

В. Что делает функция mean_and_var?

О. Мы можем пройти функцию в пошаговом режиме, повторно выполняя следующую команду:

```
(gdb) n
18          avg2 = 0;
(gdb) n
16    meanvar mean_and_var(const double *data){
```

Простое нажатие клавиши **Enter** повторяет предыдущую команду, так что даже букву `n` вводить необязательно:

```
(gdb)
18          avg2 = 0;
(gdb)
20        size_t count= 0;
(gdb)
16    meanvar mean_and_var(const double *data){
(gdb)
21        for(size_t i=0; !isnan(data[i]); i++){
(gdb)
21        for(size_t i=0; !isnan(data[i]); i++){
(gdb)
22          ratio = count/(count+1);
(gdb)
26          avg += data[i]/(count +0.0);
```

Номера строк показывают, что программа выполняется не последовательно. Объясняется это тем, что на каждом шаге отладчик выполняет машинные команды, которые необязательно точно соответствуют коду на C, из которого сгенерированы. Это нормально даже в случае, когда уровень оптимизации равен нулю. Переходы могут также отражаться на переменных – их значения ненадежны до момента выполнения второй или третьей строки после той, что выбивается из общего порядка.

Существуют и другие способы пошагового выполнения, чаще всего используются команды `s`, `n`, `u`, `c` (см. таблицу ниже). Однако на такой проход по программе может уйти весь день. Мы видим, что обход массива `data` производится в цикле `for`, так давайте поставим еще одну точку останова внутри этого цикла:

```
(gdb) b 25
Breakpoint 2 at 0x400875: file stddev_bugged.c, line 25.
```

Теперь у нас есть две точки останова, их можно посмотреть командой info break (GDB) или break list (LLDB):

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400820 in mean_and_var
                                                   at stddev_bugged.c:16
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0000000000400875 in mean_and_var
                                                   at stddev_bugged.c:25
```

Точка останова в начале функции mean_and_var нам больше не нужна, поэтому деактивируем ее [LLDB: break dis 1]:

```
(gdb) dis 1
```

После этого в столбце Enb таблицы, печатаемой командой info break, для точки останова 1 будет стоять n. Впоследствии точку останова можно реактивировать командой enable 1 (GDB) или break enable 1 (LLDB). А если вы точно знаете, что точка останова больше не понадобится, то ее можно вообще удалить командой del 1 (GDB) или break del 1 (LLDB).

В. Какие значения имеют переменные внутри цикла?

О. Можно начать выполнение программы с самого начала командой r или продолжить с места, где мы остановились, командой c:

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at
stddev_bugged.c:25
25          avg2 *= ratio;
```

Сейчас мы остановились в строке 25 и можем просмотреть все локальные переменные [LLDB: frame variable]:

```
(gdb) info local
i = 0
avg = 0
avg2 = 0
ratio = 0
count = 1
```

Можно также проверить входные аргументы функции с помощью команды GDB info args, хотя ранее мы уже и так выводили массив data. Команда LLDB frame variable выводит как локальные переменные, так и входные аргументы.

В. Мы знаем, что выведенное среднее неправильно, а как изменяется переменная avg на каждой итерации цикла?

О. Можно было бы выполнять команду p avg при каждом попадании в точку останова, но этот процесс можно автоматизировать с помощью команды display:

```
(gdb) disp avg
1: avg = 0
```

Теперь, когда мы продолжим выполнение, отладчик будет крутиться в цикле и каждый раз в точке останова печатать текущее значение avg:

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugged.c:25
25              avg2 *= ratio;
1: avg = 0

(gdb)
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at stddev_bugged.c:25
25              avg2 *= ratio;
1: avg = 0
```

Плохой знак: в программе есть строки

```
avg *= ratio;
...
avg += data[i]/(count +0.0);
```

поэтому avg должна бы изменяться на каждой итерации, но она как была равна нулю, так и остается. Установив, что это ошибка, мы можем больше не отвлекаться на переменную avg (которая в списке команды display значится под номером 1) и отменить ее автоматическую печать командой undisp 1.

В. Чему равны переменные, используемые при вычислении avg?

О. Мы уже убедились, что с data все в порядке, а как насчет ratio и count?

```
(gdb) disp ratio
2: ratio = 0

(gdb) disp count
3: count = 3
```

Выполнив еще несколько итераций цикла, мы увидим, что count, как и положено счетчику, каждый раз увеличивается на 1, а вот ratio не изменяется:

```
(gdb) c
Breakpoint 2, mean_and_var (data=data@entry=0x7fffffffe130) at
stddev_bugged.c:25
25              avg2 *= ratio;
3: count = 4
2: ratio = 0
```

В. Где присваивается значение ratio?

О. Посмотрев код в текстовом редакторе или командой l, мы увидим, что переменной ratio присваивается значение только в строке 22:

```
ratio = count/(count+1);
```

Мы уже убедились, что count увеличивается, но что-то же в этой строке неправильно. И сейчас должно быть понятно, что именно: если count целое число, то

в выражении count/(count+1) применяется целочисленное деление и, стало быть, возвращается целое число (3/4==0), хотя должна бы выполняться операция деления чисел с плавающей точкой, хорошо нам известная со школьной скамьи (3/4==0.75). Чтобы получить правильный результат (см. раздел «Меньше приведений» на стр. 157), нужно сделать так, чтобы либо числитель, либо знаменатель был числом с плавающей точкой. Для этого достаточно заменить целую константу 1 константой с плавающей точкой 1.0:

```
ratio = count/(count+1.0);
```

Отладчик не предупредил нас об этой распространенной ошибке, но помог отыскать то место в программе, где что-то впервые пошло не так, и, безусловно, найти ошибку в одной строке проще, чем во фрагменте из 50 строк. Попутно мы получили возможность проверить разнообразные аспекты поведения программы и лучше понять порядок ее выполнения и структуру стека кадров.

Ниже приведен список наиболее употребительных команд отладчика. И в GDB, и в LLDB команд гораздо больше, но показанные ниже – это те 10%, которые используются в 90% случаев. Имена переменных по большей части взяты из программы скачивания заголовков газеты «Нью-Йорк таймс», описанной в разделе «libxml и cURL» ниже.

Таблица 2.1 ❖

Группа	Команда	Назначение
Запуск	run	Запустить программу с начала
	run args	Запустить программу с начала с указанными аргументами в командной строке
Останов	b get_rss	Приостановить выполнение в начале функции
	b nyt_feeds.c:105	Приостановить выполнение перед указанной строкой
	break 105	То же, что b nyt_feeds.c:105, если мы уже остановились в файле nyt_feeds.c
	info break [GDB] break break [LLDB]	Вывести список точек останова
	watch curl [GDB] watch set var curl [LLDB]	Остановиться, если значение указанной переменной изменилось
	dis 3 / ena 3 / del 3 [GDB] break dis 3 / break ena 3 / break del 3 [LLDB]	Деактивировать/реактивировать/удалить точку останова 3. Если точек останова много, то можно деактивировать все командой disable без параметров, а затем активировать одну-две, нужные в данный момент; то же относится к командам enable и delete
Просмотр переменных	p url	Напечатать значение переменной url. Можно задать любое выражение, в том числе содержащее вызов функции
	p *an_array@10 [GDB]	Напечатать первые десять элементов массива an_array. Для печати следующих десяти элементов выполните команду *(an_array+10)@10

Таблица 2.1 ❖ *(окончание)*

Группа	Команда	Назначение
	mem read -tdouble -c10 an_array	Прочитать 10 элементов типа double из массива an_array. Для чтения следующих десяти элементов выполните команду mem read -tdouble -c10 an array+10
	info args / info vars [GDB] frame var [LLDB]	Получить значения аргументов функции или всех локальных переменных
	disp url	Печатать значение url при каждом останове программы
	undisp 3	Прекратить печать переменной с порядковым номером 3. GDB: если номер не указан, перестают печататься значения всех переменных
Потоки	info thread [GDB] thread list [LLDB]	Вывести список активных потоков
	thread 2 [GDB] thread select 2 [LLDB]	Переключиться на поток 2
Кадры	bt	Вывести стек кадров
	f 3	Показать кадр 3
	up / down	Перейти на кадр с номером, на единицу большим или меньшим текущего
Пошаговое выполнение	s	Шаг на одну строку, даже если она находится в другой функции
	n	Перейти к следующей строке, но не заходить внутрь функции
	u	Вперед до следующей строки, считая от текущей (поэтому на повторных итерациях текущего цикла останова не происходит, программа останавливается на строке, следующей за циклом)
	c	Продолжить до следующей точки останова или до завершения программы
	ret или ret 3 [GDB]	Немедленно выйти из текущей функции, вернув указанное значение (если оно задано)
	j 105 [GDB]	Перейти к произвольной (с разумными ограничениями) строке
Просмотр кода	l	Напечатать 10 строк, окружающих текущую
Повтор	Клавиша **Enter**	Нажатие клавиши **Enter** без ввода каких-либо данных приводит к повтору последней команды, что упрощает пошаговое выполнение. Если последней командой была l, то печатаются следующие 10 строк
Компиляция	make [GDB]	Запустить make, не выходя из GDB. Можно также указать цель, например: make myprog
Справка	help	Посмотреть, что предлагает отладчик

Переменные GDB

В этом разделе мы рассмотрим некоторые полезные возможности отладчика, позволяющие просматривать данные с максимальными удобствами. Все описываемые команды выполняются из командной строки; встроенные в IDE отладчики на основе GDB часто предлагают средства для встраивания их в собственный интерфейс.

Ниже приведен пример программы, которая не делает ничего полезного, но в ней есть переменная, которую можно опросить. Поскольку это программа-пустышка, не забудьте при компиляции задать флаг отключения оптимизации -O0, иначе от переменной x не останется никаких следов.

```
int main(){
    int x[20] = {};
    x[0] = 3;
}
```

Первый совет покажется новым только тем, кто не читал руководства по GDB [Stallman 2002], и есть подозрение, что это вы все. Чтобы меньше нажимать клавиши, можно завести вспомогательные переменные. Например, чтобы опросить элемент, до которого нужно долго добираться по цепочке структур, можно поступить следующим образом:

```
(gdb) set $vd = my_model->dataset->vector->data
p *$vd@10

(lldb) p double *$vd = my_model->dataset->vector->data
mem read -tdouble -c10 $vd
```

В первой строке создается вспомогательная переменная, вместо которой подставляется длинный путь. Как и в оболочке, переменные обозначаются знаком доллара. Но, в отличие от оболочки, в GDB при первом определении переменной используются команда set и знак доллара, а в LLDB – синтаксический анализатор Clang для вычисления выражений, поэтому объявление в LLDB синтаксически ничем не отличается от обычного объявления в C. Во второй строке в обоих случаях демонстрируется пример использования. Здесь мы несильно сэкономили на количестве ударов по клавишам, но если вы подозреваете некую переменную в ошибке, то наличие у нее короткого имени ускорит ввод команд опроса.

Но это не просто имена, а настоящие переменные, которые можно изменять. Остановившись в третьей или четвертой строке этой программы-пустышки, попробуйте ввести такие команды:

```
(gdb) set $ptr=&x[3]
p *$ptr = 8
p *($ptr++)  # напечатать то, на что ведет указатель, и продвинуться на следующий элемент

(lldb) p int *$ptr = &x[3]
p *$ptr = 8
p *($ptr++)
```

Команда во второй строке изменяет значение по указанному адресу. Увеличение указателя на единицу сдвигает указатель на следующий элемент списка (как описано в разделе «Все, что нужно знать об арифметике указателей» на стр. 148), поэтому после выполнения третьей строки `$ptr` указывает на `x[4]`.

Последняя форма особенно удобна, потому что нажатие клавиши **Enter** без ввода данных повторяет последнюю команду. Поскольку указатель продвигается вперед, при каждом нажатии **Enter** мы будем получать новое значение, пока не переберем всего массива. Это полезно также при обходе связанного списка. Представьте, что имеется функция `show_structure`, которая отображает элемент связанного списка и устанавливает переменную `$list` равной текущему элементу. Пусть также указатель на начало списка хранится в переменной `list_head`. Если выполнить команду

```
p $list=list_head
show_structure $list->next
```

а затем просто нажимать **Enter**, то мы обойдем весь список. Ниже мы воплотим в жизнь идею воображаемой функции для показа структуры данных.

Но сначала рассмотрим еще один прием работы с `$`-переменными. Позволю себе скопировать пару строк из сеанса работы с отладчиком на другом экране:

```
(gdb|lldb) p x+3
$17 = (int *) 0xbffff9a4
```

Обратите внимание, что результат, выведенный командой печати, начинается с `$17`. На самом деле каждый выведенный результат присваивается переменной, которую можно использовать как любую другую:

```
(gdb|lldb) p *$17
$18 = 8
(gdb|lldb) p *$17+20
$19 = 28
```

Более того, в GDB переменной с именем `$` присваивается последний выведенный результат. Поэтому если вы получили некий шестнадцатеричный адрес, то для печати хранящегося по этому адресу значения нужно просто выполнить команду `p *$`. Таким образом, показанные выше шаги можно было записать так:

```
(gdb) p x+3
$20 = (int *) 0xbffff9a4
(gdb) p *$
$21 = 8
(gdb) p $+20
$22 = 28
```

Распечатка структур

Можно определять простые макросы, что особенно полезно для отображения нетривиальных структур данных – а именно ради этого мы чаще всего и работаем с отладчиком. Даже простой двумерный массив режет глаз, если вывести его в виде длинной строки чисел. В идеальном мире для каждой структуры данных

существовала бы отдельная команда отладчика, позволяющая распечатать ее в наиболее удобном виде (или разных видах).

Но, быть может, у вас уже есть C-функция, которая распечатывает сложную структуру, встречающуюся в программе. Тогда можно написать макрос, который просто вызовет эту функцию. Средство довольно примитивное, но удобное.

В отладчике невозможно воспользоваться макросами препроцессора C, потому что они расширяются задолго до того, как отладчик видит вашу программу. Поэтому если в программе имеются полезные макросы, то их придется повторно реализовать в отладчике.

Ниже показана функция GDB, которую можно испытать, поставив точку останова в том месте функции `parse`, описанной в разделе «libxml и cURL» на стр. 334, где имеется структура `doc`, представляющая дерево XML-документа. Поместите следующие макросы в файл `.gdbinit`.

```
define pxml
   p xmlElemDump(stdout, $arg0, xmlDocGetRootElement($arg0))
end
document pxml
Распечатать на экране дерево уже открытого XML-документа (к примеру, xmlDocPtr). Скорее всего, распечатка займет несколько страниц.
Например, если дано: xmlDocPtr doc = xmlParseFile(infile);
то следует написать: pxml doc
end
```

Обратите внимание, что документация находится сразу после самой функции; просмотреть ее можно с помощью команды `help pxml` или `help user-defined`. Макрос экономит всего несколько нажатий клавиш, но поскольку работа с отладчиком сводится в основном к просмотру данных, эта экономия суммируется.

Варианты этих макросов для LLDB я покажу позже.

В библиотеке GLib имеется структура связанного списка, поэтому хотелось бы иметь средство просмотра такого списка. В примере 2.1 оно реализовано в виде двух доступных пользователю макросов (`phead` для просмотра начала списка и `pnext` для перехода к следующему элементу) и одного макроса, которого пользователь вызывать не должен (`plistdata` – для устранения избыточности при реализации `phead` и `pnext`).

Пример 2.1 ❖ Набор макросов для отображения связанного списка в GDB – пожалуй, ничего сложнее этого отладочного макроса вам никогда не понадобится (gdb_showlist)

```
define phead
   set $ptr = $arg1
   plistdata $arg0
end
document phead
Напечатать первый элемент списка. Если имеется объявление
   Glist *datalist;
   g_list_add(datalist, "Hello");
то для просмотра списка можно использовать такие команды:
```

```
gdb> phead char datalist
gdb> pnext char
gdb> pnext char
В этом макросе $ptr - указатель на текущий элемент списка, а $pdata -
данные, хранящиеся в этом элементе.
end

define pnext
   set $ptr = $ptr->next
   plistdata $arg0
end
document pnext
Сначала необходимо вызвать phead; при этом устанавливается $ptr.
Этот макрос переходит к следующему элементу списка и показывает хранящееся
в нем значение. Единственным аргументом должен быть тип данных в списке.
В этом макросе $ptr - указатель на текущий элемент списка, а $pdata -
данные, хранящиеся в этом элементе.
end

define plistdata
   if $ptr
      set $pdata = $ptr->data
   else
      set $pdata= 0
   end
   if $pdata
      p ($arg0*)$pdata
   else
      p "NULL"
   end
end
document plistdata
Предназначен для вызова из phead и pnext, см. выше. Устанавливает
переменную $pdata и печатает ее значение.
end
```

В примере 2.2 приведен пример кода, в котором в списке GList хранятся данные типа char *s. Можете поставить точки останова перед строкой 8 и после строки 9 и вызвать в них показанные выше макросы.

Пример 2.2 ❖ Пример кода для экспериментов с отладкой. Можно рассматривать как сверхкраткое введение в связанные списки из библиотеки GLib (glist.c)

```
#include <stdio.h>
#include <glib.h>

GList *list;

int main(){
    list = g_list_append(list, "a");
    list = g_list_append(list, "b");
    list = g_list_append(list, "c");

    for ( ; list!= NULL; list=list->next)
        printf("%s\n", (char*)list->data);
}
```



Можно определить функции, которые будут выполняться до или после каждого использования некоторой команды. Например, команды (GDB):

```
define hook-print
echo <----\n
end

define hookpost-print
echo ---->\n
end
```

будут выводить специального вида скобки до и после любого напечатанного значения. Самая интересная точка подключения – `hook-stop`. Команда `display` печатает значение произвольного выражения при каждом останове программы, но если вы хотите при каждом останове выполнять некоторый макрос или какую-нибудь другую команду GDB, то переопределите макрос `hook-stop`:

```
define hook-stop
pxml suspect_tree
end
```

После того как возникшие подозрения проверены, восстановите стандартное поведение:

```
define hook-stop
end
```

Для пользователей LLDB: см. `target stop-hook add`.



Ваша очередь. Макросы GDB могут включать также команду `while`, которая очень похожа на команды `if` из примера 2.1 (в начале строка вида `$ptr`, а в конце `end`). Воспользуйтесь ей, чтобы написать макрос, который выводит сразу весь список.

В LLDB все это делается немного иначе.

Во-первых, как вы, наверное, заметили, команды LLDB довольно многословны, потому что авторы ожидают, что для наиболее часто используемых команд вы сами напишете псевдонимы. Например, вот как можно было бы написать псевдонимы для команд печати массивов типа `double` или `int`:

```
(lldb) command alias dp memory read -tdouble -c%1
command alias ip memory read -tint -c%1

# Примеры использования:
dp 10 data
ip 10 idata
```

Механизм псевдонимов предназначен для сокращенной записи существующих команд. Не существует способа назначить псевдониму команды строку справки, поскольку LLDB пользуется справкой, ассоциированной с исходной командой. Для написания макросов, аналогичных показанным выше макросам GDB, в LLDB применяются регулярные выражения.

Вот версия для LLDB, которую можно поместить в файл `.lldbinit`:

```
command regex pxml
       's/(.+)/p xmlElemDump(stdout,%1, xmlDocGetRootElement(%1)0)/'
       -h "Dump the contents of an XML tree."
```

Подробное обсуждение регулярных выражений выходит за рамки этой книги (в Сети можно найти сотни пособий по регулярным выражениям). Отметим лишь,

что содержимое строки между первой и второй косой чертой будет подставлено вместо маркера%1, встречающегося между второй и третьей косой чертой.

Профилирование

Как бы быстро ни работала программа, всегда хочется, чтобы она была еще быстрее. В большинстве языков сразу дают совет: переписывайте все на C. Но мы-то и так уже пишем на C. Следующий шаг – найти функции, на которые уходит больше всего времени, их оптимизация даст наибольший эффект.

Прежде всего включите в переменную `CFLAGS` для gcc или icc флаг `-pg` (действие этого флага зависит от компилятора; gcc подготовит программу для работы с gprof, а компилятор Intel – для работы с prof, в остальном порядок действий аналогичен, поэтому я ограничусь только деталями для gcc). Программа, откомпилированная с этим флагом, будет приостанавливаться каждые несколько микросекунд и смотреть, в какой функции она сейчас находится. Эта информация записывается в двоичном формате в файл `gmon.out`.

Профилируется только сам исполняемый файл, но не скомпонованные с ним библиотеки. Поэтому если необходимо профилировать также библиотеку при исполнении тестовой программы, то придется скопировать код программы и библиотеки в одно место, а затем перекомпилировать их и собрать в один большой исполняемый файл.

После прогона программы выполните команду `gprof your_program > profile` (или `prof …`), затем откройте файл `profile` в текстовом редакторе, вы увидите список функций с указанием места, откуда они вызывались, и доли времени, проведенной программой в каждой функции. Возможно, информация о том, где в программе узкие места, станет для вас неожиданностью.

Использование Valgrind для поиска ошибок

При работе с отладчиком большая часть времени уходит на поиск места, где программа впервые начинает вести себя подозрительно. Хорошая система сама найдет это место. Иными словами, в хорошей системе программа быстро «грохнется».

Язык C в этом отношении противоречив. В некоторых языках опечатка вида `conut=15` просто приведет к созданию новой переменной, не имеющей ничего общего с переменной `count`, которую вы имели в виду; в C это будет обнаружено на этапе компиляции. С другой стороны, C позволит присвоить значение десятому элементу массива, содержащего всего 9 элементов, и долго еще будет радостно работать, пока вы не обнаружите, что там, где, по вашему мнению, должен был находиться элемент 10, на самом деле мусор.

Подобные проблемы с управлением памятью вызывают много хлопот, и потому существуют инструменты для борьбы с ними. И на одном из первых мест стоит Valgrind. Это средство перенесено на большую часть POSIX-совместимых систем (включая OS X), и установить его можно с помощью менеджера пакетов. А пользователи Windows могут поэкспериментировать с программой Dr. Memory.

Valgrind запускает виртуальную машину, которая лучше следит за использованием памяти, чем реальная, и потому знает, что вы обратились к десятому элементу массива, в котором всего 9 элементов.

Откомпилировав программу (в gcc и Clang, разумеется, с флагом `-g` для включения отладочных символов), выполните команду:

```
valgrind your_program
```

При наличии ошибки Valgrind выведет две обратные трассировки, очень похожие на те, что выводит отладчик. Первая указывает на место, где впервые было обнаружено некорректное использование, а второе – гипотеза Valgrind о том, в какой строке находится предложение, послужившее причиной ошибки, например где было произведено двойное освобождение одной и той же области памяти или где находится ближайшая операция выделения памяти с помощью `malloc`. Ошибки нередко бывают весьма тонкими, но информация о том, на какую строку обратить внимание в первую очередь, сильно сокращает время их поиска. Valgrind активно развивается – программисты ничего так не любят, как писать инструменты разработки, – я с удовольствием наблюдаю, насколько более информативными стали со временем отчеты, и ожидаю дальнейших улучшений в будущем.

Чтобы вы могли составить представление об обратной трассировке Valgrind, я внес ошибку в код примера 9.1, дважды повторив строку 14, `free(cmd)`, в результате чего память, на которую указывает `cmd`, освобождена дважды.

```
Invalid free() / delete / delete[] / realloc()
  at 0x4A079AE: free (vg_replace_malloc.c:427)
  by 0x40084B: get_strings (sadstrings.c:15)
  by 0x40086B: main (sadstrings.c:19)
Address 0x4c3b090 is 0 bytes inside a block of size 19 free'd
  at 0x4A079AE: free (vg_replace_malloc.c:427)
  by 0x40083F: get_strings (sadstrings.c:14)
  by 0x40086B: main (sadstrings.c:19)
```

Верхний кадр в обеих трассах – это библиотечный код освобождения памяти, но мы уверены, что стандартная библиотека отлажена хорошо. Перенеся внимание на ту часть стека, которая относится к написанному мной коду, я вижу, что трасса указывает на строки 14 и 15 в файле `sadstrings.c`, где действительно находятся оба обращения к `free(cmd)`.



Valgrind отлично находит условные переходы, зависящие от неинициализированных значений. Попробуйте вставить в свою программу такой код:

```
if(suspect_var) printf(" ");
```

и посмотрите, что Valgrind скажет о переменной `suspect_var`.

Можно также при обнаружении первой ошибки запустить отладчик:

```
valgrind --db-attach=yes your_program
```

В этом случае при обнаружении каждой ошибки программа будет спрашивать, хотите ли вы запускать отладчик, после чего вы сможете как обычно проверить значения подозрительных переменных.

Valgrind умеет также обнаруживать утечки памяти:

```
valgrind --leak-check=full your_program
```

Работа в подобном режиме обычно медленнее, поэтому так запускать программу каждый раз не стоит. По завершении вы получите обратную трассировку для каждого места, в котором была выделена память, впоследствии не освобожденная.

В некоторых ситуациях охота за утечками может занять очень много времени. Утечка в библиотечной функции, вызываемой миллион раз в цикле, или в программе, которая должна безостановочно работать месяцами, в конечном итоге создаст серьезные проблемы пользователям. Впрочем, нетрудно отыскать программы, которые считаются вполне надежными (на моей машине это doxygen, git, TeX, vi и многие другие), и при этом Valgrind сообщает об утечках в несколько килобайтов. В таких случаях стоит вспомнить философскую загадку о том, слышен ли звук падающего дерева в лесу, когда рядом никого нет: если ошибка не приводит к неправильным результатам и не вызывает заметного пользователю замедления, то стоит ли тратить время на ее исправление?

Автономное тестирование

Нет сомнений, что вы пишете тесты для своего кода. Для проверки небольших компонентов вы пишете *автономные тесты*, а чтобы убедиться в правильности взаимодействия разных компонентов – *интеграционные тесты*. Возможно, вы даже из тех разработчиков, которые сначала пишут автономные тесты, а только потом программу, для которой эти тесты должны проходить.

Но необходимо каким-то образом организовать все эти тесты, и тут на помощь приходит *тестовая оснастка*. Так называется система, которая подготавливает окружение для каждого теста, прогоняет его и сообщает, совпал ли полученный результат с ожидаемым. Как и в случае отладчика, я полагаю, что среди читателей найдутся как те, кто искренне удивится: разве есть люди, не пользующиеся тестовой оснасткой? – так и те, кто никогда ни о чем таком не задумывался.

Выбор велик. Несложно написать макрос-другой, которые будут вызывать тестируемую функцию и сравнивать возвращенное значение с ожидаемым. И нет недостатка в авторах, которые превратили эту простую идею в полноценную тестовую оснастку. Приведу цитату из книги [Page 2008]: «Во внутреннем репозитории Майкрософт в категории *тестовые оснастки* находится более 40 разделяемых инструментальных средств». Чтобы быть последовательным, я познакомлю вас с тестовой оснасткой из библиотеки GLib, но поскольку все они похожи и поскольку я не собираюсь приводить здесь все руководство по GLib, то рассмотрю лишь те аспекты, которым есть очевидные аналоги и в других тестовых оснастках.

У тестовой оснастки есть несколько особенностей, выгодно отличающих ее от написанного на коленке тестового макроса.

- Необходимо тестировать отказы. Если функция должна аварийно завершить программу с сообщением об ошибке, то должно быть средство проверить, что программа действительно завершилась в полном соответствии с ожиданиями.
- Все тесты должны быть изолированы, то есть прогон теста 3 не должен влиять на результат теста 4. Если вы хотите удостовериться, что две процедуры взаимодействуют правильно, сначала протестируйте их по отдельности, а затем выполните одну за другой в составе интеграционного теста.

- Перед прогоном тестов, скорее всего, нужно будет подготовить какие-то структуры данных. Подготовка окружения для теста иногда требует довольно много работы, поэтому хорошо бы иметь возможность прогнать несколько тестов, требующих одинаковой настройки.

В примере 2.3 показано несколько простых автономных тестов для словаря из раздела «Реализация словаря» ниже. Они иллюстрируют все три описанных свойства тестовой оснастки. Видно, как последний пункт определяет поток выполнения: в начале программы определяется новый тип структуры, затем идут функции для инициализации и очистки структуры этого типа, а при наличии всего этого уже нетрудно написать несколько тестов, работающих в подготовленном окружении.

Словарь – это просто множество пар ключ/значение, поэтому тестирование в основном сводится к поиску значения для заданного ключа и проверке правильности результата. Отметим, что ключ NULL недопустим, поэтому мы проверяем, что при попытке передать такой ключ программа завершается.

Пример 2.3 ❖ Тестирование словаря из раздела «Реализация словаря» (dict_test.c)

```
#include <glib.h>
#include "dict.h"

typedef struct {
   dictionary *dd;                                              ❶
} dfixture;

void dict_setup(dfixture *df, gconstpointer test_data){        ❷
   df->dd = dictionary_new();
   dictionary_add(df->dd, "key1", "val1");
   dictionary_add(df->dd, "key2", NULL);
}

void dict_teardown(dfixture *df, gconstpointer test_data){
   dictionary_free(df->dd);
}

void check_keys(dictionary const *d){                           ❸
   char *got_it = dictionary_find(d, "xx");
   g_assert(got_it == dictionary_not_found);
   got_it = dictionary_find(d, "key1");
   g_assert_cmpstr(got_it, ==, "val1");
   got_it = dictionary_find(d, "key2");
   g_assert_cmpstr(got_it, ==, NULL);
}

void test_new(dfixture *df, gconstpointer ignored){
   check_keys(df->dd);
}

void test_copy(dfixture *df, gconstpointer ignored){
   dictionary *cp = dictionary_copy(df->dd);
```

```
   check_keys(cp);
   dictionary_free(cp);
}

void test_failure(){                                          ❹
   if (g_test_trap_fork(0, G_TEST_TRAP_SILENCE_STDOUT |
                           G_TEST_TRAP_SILENCE_STDERR)){
      dictionary *dd = dictionary_new();
      dictionary_add(dd, NULL, "blank");
   }
   g_test_trap_assert_failed();
   g_test_trap_assert_stderr("NULL is not a valid key.\n");
}

int main(int argc, char **argv){
   g_test_init(&argc, &argv, NULL);
   g_test_add ("/set1/new test", dfixture, NULL,              ❺
               dict_setup, test_new, dict_teardown);
   g_test_add ("/set1/copy test", dfixture, NULL,
               dict_setup, test_copy, dict_teardown);
   g_test_add_func ("/set2/fail test", test_failure);         ❻
   return g_test_run();
}
```

❶ Элементы, используемые в наборе тестов, называются *фикстурой*. GLib требует, чтобы все фикстуры были структурами, поэтому мы создали одноразовую структуру, которая передается из функции инициализации в сам тест и затем в функцию очистки.

❷ Это функции инициализации и очистки, в которых соответственно создается и уничтожается структура данных, используемая в нескольких тестах.

❸ Сами тесты – это просто последовательности простых операций над структурами, определенными в фикстуре, и утверждения относительно ожидаемого результата этих операций. Тестовая оснастка из GLib предоставляет ряд макросов утверждений, например использованный здесь макрос для сравнения строк.

❹ Для тестирования отказа GLib пользуется определенным в POSIX системным вызовом fork (следовательно, в Windows без подсистемы POSIX это работать не будет). Вызов fork создает новый процесс, в котором исполняются предложения, перечисленные в ветви if, которые должны привести к вызову abort. Программа наблюдает за порожденным процессом и проверяет, что он действительно завершился аварийно, записав в stderr ожидаемое сообщение.

❺ Тесты собраны в наборы, идентифицируемые строками, похожими на пути. В качестве указателя на дополнительные данные (помимо инициализированных системой), которые могут использоваться в тесте, в данном случае передается NULL. Обратите внимание, что в функциях test_new и test_copy используются одинаковые функции инициализации и очистки.

❻ Если тест не требует инициализации и (или) очистки, для его прогона можно воспользоваться более простой формой.

Использование программы в качестве библиотеки

Единственная разница между библиотечной функцией и программой – это то, что в программе имеется функция main, являющаяся точкой входа, в которой начинается выполнение.

Но бывает так, что файл делает какую-то одну сравнительно несложную вещь и не заслуживает оформления в виде отдельной разделяемой библиотеки. Однако тесты для него все равно написать нужно, но я могу поместить их в тот же файл, окружив директивами условной компиляции. В примере ниже, если символ Test_operations определен (любым из рассматриваемых ниже способов), код становится программой, прогоняющей тесты, а в противном случае (типичном) этот код компилируется без main и, стало быть, является библиотекой, которую можно использовать в других программах.

```
int operation_one(){
  ...
}

int operation_two(){
  ...
}

#ifdef Test_operations

  void optest(){
    ...
  }

  int main(int argc, char **argv){
    g_test_init(&argc, &argv, NULL);
    g_test_add_func ("/set/a test", test_failure);
  }
#endif
```

Существует несколько способов определить символ Test_operations. Можно задать дополнительный флаг в файле makefile:

```
CFLAGS=-DTest_operations
```

Флаг компилятора -D, определенный в стандарте POSIX, эквивалентен включению директивы #define Test_operations в начало компилируемого *c*-файла.

При рассмотрении Automake в главе 3 мы познакомимся с оператором +=, который позволяет следующим образом добавить флаг -D в набор обычных флагов в переменной AM_CFLAGS:

```
check_CFLAGS = $(AM_CFLAGS)
check_CFLAGS += -DTest_operations
```

Условное включение main оказывается полезным и в других случаях. Например, мне часто приходится выполнять анализ причудливых наборов данных. Перед тем как произвести окончательный анализ, я должен написать функцию, которая

будет читать и очищать данные, а затем несколько функций для вывода итоговой статистики, контроля разумности данных и уведомления о ходе работы. Все это я помещаю в файл *modelone.c*. На следующей неделе у меня может возникнуть идея о новой дескриптивной модели, в которой, естественно, будут активно использоваться существующие функции очистки данных и отображения итоговой статистики. Путем условного включения функции `main` в *modelone.c* я могу быстро превратить программу в библиотеку. Вот заготовка файла *modelone.c*:

```
void read_data(){
    [здесь должна быть работа с базой данных]
}

#ifndef MODELONE_LIB
int main(){
    read_data();
    ...
}
#endif
```

Я использую `#ifndef` вместо `#ifdef`, потому что *modelone.c*, как правило, используется в качестве программы, но в остальном все работает так же, как в случае условного включения `main` для тестирования.

Покрытие

Каково тестовое покрытие вашего кода? Существуют ли в нем строки, которые не выполнялись при прогоне тестов? Вместе с gcc поставляется программа gcov, которая подсчитывает, сколько раз выполнялась каждая строка во время работы программы. Процедура выглядит следующим образом:

- Добавить в переменную `CFLAGS` для gcc флаги `-fprofile-arcs -ftest-coverage`. Стоит также добавить флаг `-O0`, чтобы оптимизатор не исключал строки.
- В ходе работы программы для каждого исходного файла *yourcode.c* порождается один или два файла: *yourcode.gcda* и *yourcode.gcno*.
- Команда `gcov` *yourcode.gcda* выводит в `stdout` процентную долю исполняемых строк, которые действительно исполнялись во время прогона программы (объявления, директивы `#include` и т. п. не считаются), и создает файл *yourcode.c.cov*.
- В первом столбце таблицы в файле *yourcode.c.cov* показано, сколько раз каждая исполняемая строка исполнялась тестами, а строки, которые не исполнялись ни разу, помечены большим жирным маркером `#####`. Именно на них следует обращать внимание при написании очередного теста.

В примере 2.4 приведен скрипт оболочки, содержащий все описанные выше шаги. Я воспользовался встроенным документом для генерации makefile. После компиляции, прогона и выполнения `gcov` я с помощью `grep` ищу маркеры `#####`. Флаг `-C3` при вызове GNU `grep` означает, что нужно выводить три контекстные строки, окружающие те, где найдены совпадения. Этот флаг не описан в стандарте POSIX, как, впрочем, и `pkg-config` и флаги для генерации тестового покрытия.

Пример 2.4 ❖ Скрипт, который компилирует программу для генерации тестового покрытия, прогоняет тесты и ищет строки, еще не покрытые тестами (gcov.sh)

```
cat > makefile << '------'
P=dict_test
objects= keyval.o dict.o
CFLAGS = `pkg-config --cflags glib-2.0` -g -Wall -std=gnu99 \
         -O0 -fprofile-arcs -ftest-coverage
LDLIBS = `pkg-config --libs glib-2.0`
CC=gcc

$(P):$(objects)
------

make
./dict_test
for i in *gcda; do gcov $i; done;
grep -C3 '#####' *.c.gcov
```

Встроенная документация

Документация необходима. Вы это знаете, как знаете и то, что при изменении кода нужно поддерживать документацию в актуальном состоянии. И тем не менее зачастую документация – первое, что откладывают на потом. Ведь так легко сказать: *работает же, а документацию напишу позже*.

Поэтому необходимо упростить написание документации по максимуму. А это означает, что документация должна быть в том же файле, что сам документируемый код, как можно ближе к нему, и при этом нам необходимы средства извлечь документацию из файла с кодом.

Размещение документации рядом с кодом означает также, что вы с большей вероятностью прочтете ее. Нужно выработать у себя привычку перечитывать документацию по функции, перед тем как вносить в нее какие-либо модификации. Это не только позволит вспомнить, что происходит внутри функции, но и понять, нужно ли будет изменить документацию после изменения кода.

Я расскажу о двух способах встраивания документации в код: Doxygen и CWEB. То и другое легко установить с помощью менеджера пакетов.

Doxygen

Doxygen – простая система с простыми целями. Она рассчитана на то, что автор снабдит описанием каждую функцию, структуру и другие подобные элементы программы. Предполагается, что таким образом документируется внешний интерфейс для пользователей, которые никогда не будут заглядывать в сам код. Описание размещается в блоке комментария непосредственно над функцией, структурой и т. п., так чтобы было естественно сначала написать документацию, а потом саму функцию – не нарушая данных по поводу ее работы обещаний.

Синтаксис Doxygen несложен, приведенных ниже правил достаточно, чтобы начать им пользоваться.

- Если блок комментария начинается косой чертой с двумя звездочками – `/** вот так */`, то Doxygen его разбирает. Комментарии вида `/* вот так */` игнорируются.
- Если вы хотите, чтобы Doxygen обработал файл, то нужно поставить комментарий `/** \file */` в начале файла; см. пример. Если вы забудете это сделать, то Doxygen ничего не сгенерирует для этого файла и не предупредит о возможной ошибке.
- Размещайте комментарий непосредственно перед функцией, структурой и т. д.
- Описания функций могут (и должны) включать блоки `\param`, в которых описываются входные параметры, и блок `\return` с описанием возвращаемого значения. См. пример ниже.
- Используйте блок `\ref` для перекрестных ссылок на другие документированные элементы (в том числе функции или страницы).
- Вместо обратной косой черты можно использовать знак `@`, например: `@file`, `@mainpage` и т. д. Этот синтаксис имитирует систему JavaDoc, которая, в свою очередь, похоже, имитирует WEB. Будучи пользователем LaTeX, я предпочитаю обратную косую черту.

Для запуска Doxygen необходим конфигурационный файл, а настраиваемых параметров более чем достаточно. В Doxygen применяется ловкий трюк, чтобы облегчить работу; выполните команду

```
doxygen -g
```

и конфигурационный файл будет сгенерирован автоматически. Потом можете его открыть и отредактировать; разумеется, он прекрасно документирован. После этого запускайте саму программу `doxygen`, чтобы сгенерировать документацию в формате HTML, PDF, XML или страниц руководства – как указано в конфигурационном файле.

Если на машине установлен пакет Graphviz (менеджер пакетов вам в помощь), то Doxygen может сгенерировать *графы вызовов*: диаграммы со стрелками, показывающими, как одни функции вызывают другие. Если вас попросят быстро разобраться в запутанной программе, то это неплохой способ понять, как она устроена.

Я с помощью Doxygen документировал программу, описанную в разделе «libxml и cURL» ниже; посмотрите, как выглядит снабженный комментариями код, или запустите Doxygen и ознакомьтесь с порожденной документацией.

Все встречающиеся в этой книге фрагменты, начинающиеся с `/**`, записаны в формате Doxygen.

Повествование

Документация должна состоять по меньшей мере из двух частей: техническая, в которой приведена детальная информация обо всех функциях, и повествовательная, в которой объясняется, для чего предназначен пакет и как в нем сориентироваться.

Повествовательную часть начинайте в блоке комментария с заголовком \mainpage. Если генерируется документация в формате HTML, то эта часть станет файлом *index.html* сайта – первой страницей, которую увидит посетитель. На ней могут быть размещены ссылки на другие страницы, каждая из которых должна иметь заголовок вида:

```
/** \page onewordtag Заголовок вашей страницы
*/
```

На главной странице (или на любой другой, в том числе на странице документации по функции) поместите блок \ref onewordtag, который преобразуется в ссылку на страницу. Если есть необходимость, можно пометить и озаглавить также главную страницу.

Повествовательные страницы могут находиться в любом месте кода. Иногда имеет смысл приблизить текст к коду, а иногда вынести в отдельный файл, состоящий исключительно из блоков комментариев для Doxygen; назвать его можно, например, *documentation.h*.

Грамотное программирование с помощью CWEB

Систему форматирования документов TeX часто приводят в качестве эталона сложной системы, спроектированной так, как надо. Ей уже исполнилось 35 лет, но до сих пор она (по мнению автора) порождает самые красиво отформатированные математические тексты для любой существующей наборной системы. Многие системы, разработанные позднее, даже не пытаются конкурировать с TeX, а используют ее в качестве основы для типографского набора. Автор, Дональд Кнут, предлагал вознаграждение за найденные ошибки, но в конце концов снял свое предложение, поскольку за вознаграждением за много лет так никто и не обратился.

Д-р Кнут объясняет высочайшее качество TeX тем, как она написана. Он употребляет термин *грамотное программирование* (literate programming), понимая под этим тот факт, что каждому процедурному блоку предшествует объяснение (на понятном английском языке) его назначения и функционирования. Конечный продукт выглядит как неформальное описание кода, в которое тут и там вкраплен сам код, необходимый, чтобы формализовать это описание для компьютера (в противоположность типичной документированной программе, в которой кода гораздо больше, чем объяснений). Кнут писал TeX, пользуясь системой WEB, которая вплетает в пояснительный англоязычный текст код на языке PASCAL. В наши дни код будет написан на C, и теперь, когда к созданию красивой документации привлечен TeX, мы вполне можем использовать его в качестве языка разметки пояснительной части. Итак, познакомимся с CWEB.

Что касается вывода результатов, то легко найти учебники, разъясняющие, как использовать CWEB для организации и даже презентации содержимого (например, [Hanson 1996]). Если кто-то другой захочет изучить ваш код (скажем, коллега или член группы критического анализа), то CWEB сможет оказать реальную помощь.

Программу в разделе «Пример: основанная на агентах модель формирования группы» (стр. 281) я написал с использованием CWEB; ниже приведена краткая сводка того, что нужно знать для ее компиляции и понимания специфичных для CWEB особенностей.

- Файлы CWEB принято сохранять с расширением *.w*.
- Для создания *tex*-файла выполните команду `cweave groups.w`; затем для создания PDF-файла – команду `pdftex groups.tex`.
- Для создания *c*-файла выполните команду `ctangle groups.w`. GNU make знает о расширении *.w*, поэтому `make groups` запустит `ctangle` автоматически.

Программа `ctangle` удаляет комментарии, а это значит, что CWEB и Doxygen несовместимы. Быть может, имеет смысл создать файл, в котором описаны все открытые функции и структуры, и обрабатывать его программой doxygen, оставив CWEB для основного набора файлов с кодом.

Ниже приведено руководство по CWEB, сведенное к нескольким пунктам.

- Все специальные коды CWEB представляют собой знак `@`, за которым следует один символ. Правильно писать `@<titles@>`, а не `@<incorrect titles>@`.
- В каждом сегменте имеется комментарий, за ним код. Комментарий может быть пустым, но структура комментарий–код должна выдерживаться, иначе возможны самые разные ошибки.
- Текстовая секция начинается знаком `@` с последующим пробелом. Затем пояснения в формате TeX.
- Безымянный блок кода должен начинаться с `@c`.
- Именованный блок кода начинается с заголовка, после которого должен находиться знак равенства (потому что это определение): `@<an operation@>=`.
- Этот блок копируется буквально всюду, где встречается соответствующий заголовок. Иначе говоря, имя блока – это, по существу, макрос, расширяемый заданным вами кодом, но без дополнительных возможностей макросов препроцессора C.
- Секции (в примере есть секции, относящиеся к членству в группах, инициализации, построению графиков с помощью Gnuplot и т. д) начинаются управляющей последовательностью `@*`, за которой следует заголовок, завершающийся точкой.

Этого достаточно, чтобы начать самостоятельно работать с CWEB. Изучите вышеупомянутый пример на стр. 281 — все ли там понятно?

Как стать классной машинисткой

При отборе многих тем для этой книги я руководствовался собственным опытом консультирования коллег в части программирования на C и по ходу дела выяснял, что вызывает у них трудности. Для одних камнем преткновения была настройка окружения, другим никак не давались указатели, и, как ни странно, немало было и таких, кто не мог поладить с клавиатурой. Конечно, прямого отношения к программированию это не имеет, но люди, неуверенно печатающие на клавиатуре, не склонны использовать язык, в котором такой вот изобилующий знаками препинания текст `for (i=0; i<10; i++)` — обычное дело.

И вот какой совет я даю тем, кто испытывает такого рода трудности: возьмите тонкую футболку и набросьте ее на клавиатуру. Засуньте руки под футболку и начинайте печатать.

Идея в том, чтобы нельзя было украдкой взглянуть на клавиатуру и найти нужную клавишу. Вам, конечно, известно, что клавиши бегать не умеют и всегда находятся там, где вы их оставили. Однако те краткие паузы, которые мы делаем, чтобы убедиться в этом, снижают скорость печати.

Если печать вслепую поначалу вызывает горькое разочарование, постарайтесь преодолеть его и привыкнуть к тем редко используемым клавишам, которые раньше не удосужились выучить. Обретя уверенность в обращении с клавиатурой, вы сможете уделить больше умственной энергии написанию программ.

Проверка ошибок

Любой уважающий себя учебник программирования обязан включать хотя бы одну лекцию на тему о том, как важно обрабатывать ошибки, возвращенные вызываемыми функциями.

Ну что ж, считайте, что лекцию вы прослушали. И обратимся к другой стороне проблемы: как и когда возвращать ошибки из своей функции. Существует великое множество типов ошибок, встречающихся в самых разных контекстах, поэтому рассмотрим несколько более частных вопросов.

- Что пользователь будет делать с сообщением об ошибке?
- Предназначена ошибка пользователю или другой функции?
- Как уведомить пользователя об ошибке?

Третий вопрос я отложу на потом (раздел «Возврат нескольких значений из функции» на стр. 225), но и первые два дадут нам достаточно пищи для размышлений.

Ошибки и пользователи

Бездумный подход к обработке ошибок, при котором код пестрит проверками – ведь много не бывает, да? – вовсе не обязательно является правильным. Строки, где обрабатываются ошибки, необходимо сопровождать, как и все остальные, а любой пользователь вашей функции вынес из лекций твердый урок – обрабатывай все возможные коды ошибок. Поэтому если вы возвращаете код ошибки, который невозможно разумно обработать, то у пользователя функции останется чувство вины и неуверенности в себе. Существует такая вещь, как избыток информации.

Чтобы подступиться к вопросу о том, как будет использована информация об ошибке, рассмотрим смежный вопрос: как вообще пользователь может столкнуться с ошибкой.

- *Иногда пользователь не может узнать, допустим ли входной параметр, не вызвав функцию.* Классический пример – поиск ключа в множестве ключей и значений, в результате которого обнаруживается, что ключа там нет. В таком случае можно считать, что это функция поиска, возвращающая ошибку, если ключ отсутствует в множестве. А можно подойти по-другому и считать, что у функции две задачи: искать ключ и сообщать пользователю, есть ключ или нет.

  Или возьмем пример из школьного курса алгебры: для того чтобы найти корни квадратного уравнения, необходимо вычислить квадратный корень

sqrt(b*b - 4*a*c); если выражение в скобках отрицательно, то вещественных решений у уравнения нет. Смешно ожидать, что пользователь будет вычислять, чему равно b*b - 4*a*c, чтобы понять, можно обращаться к функции или нет; разумнее сделать так, чтобы функция для решения квадратного уравнения либо возвращала его корни, либо сообщала, что вещественных корней нет.

В этих примерах проверка входных параметров нетривиальна, а «плохие» параметры – это даже не ошибка, а результат естественного использования функции. Если функция обработки ошибок аварийно завершает программу или делает еще что-то столь же деструктивное (как показанный ниже обработчик), то вызывать ее в подобных случаях не следует.

- *Пользователь передает откровенно неправильные параметры*, например нулевой указатель или еще какие-то некорректные данные. Ваша функция должна отлавливать такие вещи, чтобы предотвратить нарушение защиты памяти или что-то в этом роде, но трудно представить, что вызывающая программа могла бы сделать с информацией о таких ошибках. В документации по функции yourfn ясно сказано, что указатель не должен быть равен NULL, и если пользователь игнорирует это и пишет int* indata=NULL; yourfn(indata), а в ответ получает ошибку вида Error: NULL pointer input, то непонятно, что именно пользователь должен сделать по-другому.

  В начале функции обычно имеется несколько строк вида if (input1==NULL) return -1; ... if (input20==NULL) return -1;, и мой опыт показывает, что перечислять в документации, какое именно из базовых требований нарушено – значит впадать в грех избыточной информации.

- *Ошибка возникла по внутренним причинам.* Сюда относятся и ошибки типа «такого не должно быть», когда в результате внутреннего вычисления получается невозможный ответ – то, что в американском психоделическом рок-мюзикле «Волосы» называется «повреждениями плоти». Примерами могут служить отсутствие ответа от оборудования, пропадание сети или подключения к базе данных.

  Повреждения плоти обычно можно устранить (например, пошевелить сетевой кабель). Или если пользователь просит разместить в памяти гигабайт данных, а столько места нет, то вполне разумно будет сообщить об ошибке нехватки памяти. Однако если не получается выделить память для строки из 20 символов, то либо машина перегружена и скоро ее работа станет нестабильной, либо вообще случился пожар – в таком случае вызывающая программа вряд ли сможет сделать что-то осмысленное для восстановления. В зависимости от контекста сообщения об ошибках типа «пожар, спасайся кто может» бывают контрпродуктивными и могут расцениваться как избыток информации.

  Внутренние ошибки (то есть не связанные с внешними условиями и некорректными входными параметрами) вызывающая программа обработать не может. В таком случае детальная информация, по-видимому, является

избыточной для пользователя. Вызывающая сторона должна знать, что результат работы функции ненадежен, но наличие многочисленных кодов ошибок только усложняет работу пользователя (который все эти ошибки обязан обрабатывать).

Учет контекста, в котором работает пользователь

Как было сказано выше, мы нередко проверяем в функции правильность входных данных. Неправильные параметры – это, строго говоря, не ошибка, и функция должна бы в этих случаях возвращать осмысленное значение, а не вызывать обработчик ошибок. Далее в этом разделе мы будем рассматривать только *настоящие ошибки*.

- Если пользователь программы имеет доступ к отладчику и работает в контексте, где его использование практически возможно, то самый простой способ уведомить об ошибке – вызвать функцию `abort`, которая аварийно завершает программу. После этого пользователь может ознакомиться с локальными переменными и обратной трассировкой прямо на месте преступления. Функция `abort` входила в стандарт C с самого начала (она объявлена в заголовке `<stdlib.h>`).
- Если пользователем функции является, например, Java-программа или человек, понятия не имеющий, что такое отладчик, то вызов `abort` – прямое кощунство, а правильно было бы вернуть код ошибки, описывающий, что произошло.

Оба случая встречаются на практике, поэтому было бы разумно предоставить пользователю возможность выбрать подходящий режим работы.

Давненько уже не встречались мне нетривиальные библиотеки, в которых не было бы собственной реализации макроса для обработки ошибок. На этом уровне стандарт C ничего не предлагает, но написать такой макрос с помощью стандартных средств нетрудно, чем все и пользуются.

Стандартный макрос `assert` (совет: `#include <assert.h>`) проверяет переданное ему утверждение и останавливает программу, если оно оказывается ложным. Конкретные реализации могут быть различны, но идея такова:

```
#define assert(test) (test) ? 0 : abort();
```

Сам по себе макрос `assert` полезен, когда нужно проверить правильность выполнения промежуточных шагов внутри функции. Я также люблю применять `assert` в качестве документации: конечно, это условие, которое должен проверить компьютер, но когда человек видит строку `assert(matrix_a->size1 == matrix_b->size2)`, он понимает, что размерности двух матриц должны быть определенным образом связаны. Однако `assert` предполагает только одну реакцию (аварийное завершение), поэтому утверждения необходимо обертывать как-то иначе.

В примере 2.5 приведен макрос, удовлетворяющий обоим условиям, в разделе «Макросы с переменным числом аргументов» я рассмотрю его более подробно. Отмечу, что некоторые пользователи функций работают с stderr без затруднений, тогда как у других вообще нет средств для работы с этим потоком.

Пример 2.5 ❖ Макрос для обработки ошибок: сообщить или вывести в журнал и предоставить пользователю возможность решить, нужно ли останавливать программу или можно продолжить выполнение (*stopif.h*)

```
#include <stdio.h>
#include <stdlib.h> //abort

/** Присвоить этой переменной значение \c 's', если нужно, чтобы программа
    останавливалась после ошибки.
    В противном случае программа будет возвращать код ошибки.*/
char error_mode;

/** Куда писать сообщения об ошибках? Если \c NULL, писать в \c stderr. */
FILE *error_log;

#define Stopif(assertion, error_action, ...) {\
       if (assertion){    \
          fprintf(error_log ? error_log : stderr, __VA_ARGS__);  \
          fprintf(error_log ? error_log : stderr, "\n"); \
          if (error_mode=='s') abort(); \
          else                 {error_action;} \
       } }
```

Вот несколько примеров использования этого макроса:

```
Stopif(!inval, return -1, "inval must not be NULL");
Stopif(isnan(calced_val), goto nanval, "Calced_val was NaN. Cleaning
up, leaving.");
...
nanval:
free(scratch_space);
return NAN;
```

Самый распространенный способ обработки ошибки – просто вернуть некое значение, поэтому если вы применяете данный макрос в представленном виде, то будьте готовы часто набирать `return`. Впрочем, это, может, и неплохо. Авторы программ часто сетуют на то, что хитроумные конструкции try-catch – по существу, не более чем модернизированный вариант всеми осуждаемого `goto`. Например, во внутреннем руководстве Google по стилю кодирования рекомендуется избегать блоков try-catch, и в обоснование приводится именно это сходство с `goto`. Утверждается, что лучше явно дать понять читателю, что поток выполнения программы будет перенаправлен на выдачу ошибки (и куда именно), а методика обработки ошибки должен быть как можно проще.

Как следует возвращать уведомление об ошибке?

К этому вопросу я еще вернусь в главе о работе со структурами (точнее, в разделе «Возврат нескольких значений из функции»), потому что если уровень сложности функции превышает некий порог, то возврат структуры – вещь очень полезная, а тогда уже нетрудно и разумно добавить в структуру переменную, описывающую ошибку. Например, в предположении, что функция возвращает структуру `out`, в которой есть поле `error` типа `char*`, мы можем написать:

```
Stopif(!inval, out.error="inval must not be NULL"; return out
           , "inval must not be NULL");
```

В библиотеке GLib имеется система обработки ошибок на основе специального типа `GError`, указатель на который нужно передавать в качестве аргумента любой функции. Предлагается также несколько дополнительных средств, помимо представленного в примере 2.5 макроса, в том числе области ошибок и упрощенный метод передачи ошибок от дочерних функций родительским, – но все это ценой увеличения сложности.



Глава 3. Создание пакета для проекта

Раз вы дочитали до этого места, то уже знакомы с инструментами для решения важнейших проблем, возникающих при программировании на C, в частности отладки и документирования кода. Если вам не терпится поскорее заняться самим языком, то можете сразу перейти к части II. А в этой и в следующей главах будут рассмотрены инструменты для тяжелых работ – совместной разработки и распространения программ, а именно средства сборки пакетов и система управления версиями. Попутно мы будем говорить о том, как эти инструменты помогают программировать соло.

Я уже упоминал об этом во введении, но, поскольку введения никто не читает, повторю: сообщество, сложившееся вокруг C, привержено высочайшим стандартам интероперабельности. Да, оглядевшись вокруг в своем офисе или в кафетерии, вы обнаружите, что все пользуются примерно одними и теми же инструментами, но, выйдя за пределы узкого мирка, вы столкнетесь с поразительным разнообразием. Лично я довольно часто получаю письма от людей, которые пользуются написанным мной кодом в системах, которых я никогда не видывал; мне это кажется удивительным, и я бываю очень доволен, что не шел по легкому пути – написать код, который будет работать только на моей платформе, – а стремился к интероперабельности.

В наши дни основной технологией распространения кода является Autotools – система автоматической генерации файла makefile, идеально подходящего данной системе. Мы уже встречались с ней в разделе «Сборка библиотек из исходного кода» выше, когда с ее помощью смогли быстро установить библиотеку GNU Scientific Library. Возможно, сами вы никогда вплотную не работали с Autotools, но именно так люди, сопровождающие систему управления пакетами, создают пакеты для вашей конкретной системы.

Разобраться в том, что делает Autotools, затруднительно, не понимая, как работает makefile, поэтому для начала мы рассмотрим этот вопрос подробнее. В первом приближении файл makefile – это организованный набор команд оболочки, а значит, необходимо знать, какие средства автоматизации работы предоставляет оболочка. Путь нам предстоит долгий, но, пройдя его до конца, вы сможете:

- использовать оболочку для автоматизации работы;

- применять файлы makefile для организации последовательностей задач, выполняемых оболочкой.
- использовать Autotools, чтобы предоставить пользователям возможность автоматически генерировать файлы makefile в любой системе.

Оболочка

POSIX-совместимая оболочка должна обладать следующими свойствами:

- развитые макросредства, позволяющие заменить один текст другим, то есть предоставлять синтаксис *расширения*;
- Тьюринг-полный язык программирования;
- интерактивный интерфейс – командная строка – включающий множество удобных приемов работы;
- система запоминания и повторного использования ранее набранных команд: история;
- много других вещей, о которых я не буду здесь упоминать, в том числе управление заданиями и многочисленные встроенные утилиты.

Синтаксис оболочки *очень* богат, и в этом разделе мы сможем сорвать только немногие низко висящие плоды, иллюстрирующие вышеперечисленные пункты. Существует много оболочек (и ниже встретится врезка с предложением попробовать оболочки, отличные от подразумеваемой по умолчанию), но, если явно не оговорено противное, в этом разделе мы будем придерживаться стандарта POSIX.

Не буду тратить много времени на средства обеспечения интерактивности, но все же не могу не упомянуть одно, хотя оно и не включено в POSIX: завершение по нажатию клавиши **Tab**. В оболочке bash, если вы введете начало имени файла и нажмете **Tab**, имя будет автоматически продолжено до конца, если существует только один вариант завершения; если же вариантов несколько, то повторное нажатие **Tab** позволит увидеть их список. Если вы хотите узнать, сколько команд можно ввести в командной строке, дважды нажмите **Tab** в пустой строке, и bash выведет весь список. Другие оболочки идут гораздо дальше bash: наберите `make <tab>` в оболочке zsh, и она найдет все цели в файле makefile. Оболочка Friendly Interactive shell (fish) ищет рефераты в страницах руководства, поэтому при вводе `man l<tab>` она выведет однострочные рефераты для всех команд, начинающихся буквой L; иногда это позволяет обойтись без открытия самой страницы руководства.

Пользователей оболочек можно отнести к двум группам: те, кто вообще не знает об автоматическом завершении и прочих трюках, связанных с клавишей **Tab**, и те, кто пользуется ими *постоянно при вводе каждой команды*. Если раньше вы принадлежали к первой группе, то переход во вторую вам определенно понравится.

Замена команд оболочки их выводом

Оболочка во многом ведет себя как макроязык, который позволяет заменять одни фрагменты текста другими. В оболочке такие подстановки называются *расшире-*

ниями, и их довольно много разных; в этом разделе мы рассмотрим подстановки переменных, подстановки команд, элементы подстановок в историю, а также приведем примеры расширения тильды и арифметических подстановок для примитивного калькулятора. Оставляю вам удовольствие прочитать в руководстве о расширении псевдонимов, расширении скобок, расширении параметров, разбиении на слова, расширении путей и расширении по маске.

Подстановка переменной – это простое расширение. Если переменной следующим образом присвоено значение

```
onething="another thing"
```

в командной строке, то впоследствии можно написать:

```
echo $onething
```

и на экране будет напечатана строка `another thing`.

Оболочка требует, чтобы ни по одну сторону от знака = не было пробелов, со временем это начинает раздражать.

Когда одна программа запускает другую (в POSIX C это делается с помощью системного вызова `fork()`), дочерней программе передаются копии всех переменных окружения. Разумеется, точно так же работает и оболочка: когда вы вводите команду, оболочка запускает новый процесс и передает ему все переменные окружения.

Однако переменные окружения – это только подмножество всех переменных оболочки. Показанное выше присваивание устанавливает переменную оболочки. Если же написать

```
export onething="another thing"
```

то эта переменная еще и становится доступной для экспорта. Значения экспортируемых переменных по-прежнему можно изменять в оболочке.

Следующее расширение, которое мы рассмотрим, связано со знаком обратного апострофа `, который не следует путать с прямой одиночной кавычкой '.



Прямая одиночная кавычка (', в отличие от обратного апострофа) означает, что расширение производить не нужно. Последовательность команд

```
onething="another thing"
echo "$onething"
echo '$onething'
```

напечатает:

```
another thing
$onething
```

Обратные апострофы заменяют заключенную в них команду ее выводом.

В примере 3.1 показан скрипт, который подсчитывает, сколько строк в программе на C содержат один из символов `;`, `)` или `}`. Конечно, для большинства целей количество строк исходного кода в любом случае не годится в качестве метрики, но тем не менее этот инструмент все же не хуже любого другого, да к тому же занимает всего одну строку кода на языке оболочки.

Пример 3.1 ❖ Подсчет строк с помощью переменных оболочки и утилит, описанных в POSIX (linecount.sh)

```
# Подсчитаем число строк, содержащих ;, ) или }, назовем этот счетчик Lines.
Lines=`grep '[;)}]' *.c | wc -l`

# Теперь подсчитаем, сколько строк в списке каталогов; назовем счетчик
# Files.
Files=`ls *.c |wc -l`

echo files=$Files and lines=$Lines

# Арифметическое расширение обозначается двойными круглыми скобками.
# В bash остаток отбрасывается, подробнее об этом позже.
echo lines/file = $(($Lines/$Files))

# Эти переменные можно использовать и во встроенном документе-скрипте.
# Если установить scale=3, то результаты будут печататься с 3 знаками
# после десятичной точки.
# (Можно также воспользоваться командой bc -l (буква эль), которая
# устанавливает scale=20)
bc << ---
scale=3
$Lines/$Files
---
```

Для запуска этого скрипта нужно выполнить команду `. linecount.sh`. Точка в POSIX-совместимой оболочке означает загрузку исходного кода скрипта. Скорее всего, ваша оболочка позволяет сделать то же самое с помощью нестандартной, но куда более понятной записи `source linecount.sh`.



В командной строке пара обратных апострофов в большинстве случаев эквивалентна конструкции `$()`. Например, ``echo `date` `` – то же самое, что `echo $(date)`. Однако в `make` конструкция `$()` употребляется для других целей, поэтому внутри makefile следует пользоваться обратными апострофами.

Применение циклов for в оболочке для обработки набора файлов

Перейдем собственно к программированию – с предложениями `if` и циклами `for`.

Но сначала отмечу некоторые подводные камни и неприятные сюрпризы при написании скриптов оболочки.

- Областей видимости, по существу, нет – практически все глобально.
- Оболочка – это макроязык, поэтому все нюансы взаимодействия с текстом, о которых вас предупреждали при работе с препроцессором C (см. раздел «Выращивание устойчивых и плодоносящих макросов» на стр. 172), во многом относятся и к скриптам оболочки.
- Не существует отладчика, умеющего выполнять скрипт в пошаговом режиме, как описано в разделе «Работа с отладчиком» выше, хотя современные оболочки предоставляют средства для трассировки ошибки и выполнения скриптов с выдачей подробной информации.

- Придется привыкнуть к мелким пакостям, которые поджидают на каждом шагу; например, нельзя отделять пробелами знак = в присваивании `onething=another`, зато обязательно окружать пробелами скобки `[` и `]` в предложении `if [ -e ff ]` (поскольку на самом деле это ключевые слова, только небуквенные).

Некоторых эти детали не раздражают, и лично я ♥ оболочку. Я пишу скрипты оболочки, чтобы автоматизировать выполнение команд из командной строки, и как только код начинает чрезмерно усложняться, скажем, появляются вложенные функции, я перехожу на Perl, Python, awk – в общем, на более подходящий язык.

Для меня прелесть языка программирования, позволяющего писать код прямо в командной строке, становится особенно очевидной из-за возможности применять одну команду к нескольким файлам. Давайте сделаем резервные копии всех *c*-файлов по старинке – скопировав каждый файл в новый с расширением *.bkup*:

```
for file in *.c;
do
  cp $file ${file}.bkup;
done
```

Обратите внимание на точку с запятой: она должна находиться после списка файлов в той же строке, что ключевое слово `for`. Я специально подчеркиваю эту деталь, потому что при записи цикла в одной строке:

```
for file in *.c; do cp $file ${file}.bkup; done
```

я неизменно забываю, что нужно писать `; do`, а не `do ;`.

Цикл `for` полезен, когда нужно запустить некую программу `N` раз. Например, скрипт `benford.sh` ищет в коде на С числа, начинающиеся с определенной цифры (то есть нас интересуют цифры в начале строки или после символа, отличного от цифры), и выводит найденные строки в файл.

Пример 3.2 ❖ Для каждой цифры i ищем в тексте последовательность (не цифра)i и подсчитываем такие строки (benford.sh)

```
for i in 0 1 2 3 4 5 6 7 8 9; do grep -E '(^|[^0-9.])'$i \
*.c > lines_with_${i}; done
wc -l lines_with*   // грубая гистограмма распределения цифр
```

Проверку *закона Бенфорда* оставляю в качестве упражнения для читателя.

Фигурные скобки в `${i}` отделяют имя переменной от окружающего текста; в данном случае они необязательны, но понадобились бы, если бы нужно было записать имя файла вида `${i}lines`.

Возможно, на вашей машине имеются команды `seq` – они определены в стандарте BSD/GNU, но не в POSIX. Тогда для генерации последовательности можно было бы воспользоваться обратными апострофами:

```
for i in `seq 0 9`; do grep -E '(^|[^0-9.])'$i *.c > lines_with_${i}; done
```

Запустить программу тысячу раз? Ничего не может быть проще:

```
for i in `seq 1 1000`; do ./run_program > ${i}.out; done

#или дописывать все в конец одного файла:
for i in `seq 1 1000`; do
  echo output for run $i: >> run_outputs
  ./run_program >> run_outputs
done
```

Проверка наличия файла

Допустим, ваша программа должна читать данные из текстового файла и записывать в базу данных. Мы хотим читать данные только один раз или в виде псевдокода: if (база данных существует) then (ничего не делать) else (создать базу данных на основе текстового файла).

В командной строке мы могли бы реализовать эту идею с помощью многоцелевой команды test, которая обычно встроена в оболочку. Давайте попробуем: выполните команду ls, чтобы узнать имя какого-нибудь заведомо существующего файла, а затем с помощью test проверьте, что этот файл действительно существует:

```
test -e a_file_i_know
echo $?
```

Сама по себе команда test ничего не выводит, но, будучи программистом на C, вы знаете, что в любой программе есть функция main, которая возвращает целое число, его-то мы здесь и используем. По соглашению возвращаемое значение интерпретируется как порядковый номер проблемы, то есть 0 означает, что никаких проблем нет, а 1 в данном случае – что файл не существует (именно поэтому main по умолчанию возвращает 0, о чем мы будем говорить в разделе «Ни к чему явно возвращать значение из main» на стр. 154). Оболочка не печатает возвращенное значение на экране, а сохраняет его в переменной $?, которую можно распечатать командой echo.

У самой команды echo тоже есть возвращаемое значение, и после выполнения echo $? оно будет присвоено переменной $?. Если вы хотите несколько раз использовать значение $?, полученное после выполнения какой-то команды, то сохраните его в другой переменной: returnval=$?.

А мы проверим это значение в предложении if и будем что-то делать, только если файл не существует. Как и в C, знак ! означает *не*.

Пример 3.3 ❖ Проверка, погруженная в предложение if/then, – выполните эту команду несколько раз (. iftest.sh; . iftest.sh; . iftest.sh) и понаблюдайте, как файл то появляется, то исчезает (iftest.sh)

```
if test ! -e a_test_file; then
   echo test file had not existed
   touch a_test_file
else
   echo test file existed
   rm a_test_file
fi
```

Обратите внимание, что, как и в цикле for, точка с запятой находится не там, где казалось бы естественным (мне, по крайней мере). Еще не забудьте про замечательное правило: любой блок if должен заканчиваться словом fi. Кстати говоря, синтаксис else if недопустим, нужно писать elif.

Чтобы было проще запускать этот скрипт многократно, запихнем его в одну строку, для которой, правда, не хватает ширины страницы. Ключевые слова [и] эквивалентны слову test, поэтому, встретив такую конструкцию в чужом скрипте и желая узнать, что имелось в виду, читайте man test.

```
if [ ! -e a_test_file ]; then echo test file had not existed; ↩
touch a_test_file; else echo test file existed; rm a_test_file; fi
```

Поскольку очень много программ придерживается соглашения нуль=OK, а ненуль==проблема, то мы можем использовать предложения if без test, чтобы выразить мысль *если программа завершилась нормально, то...* . Например, нередко применяют команду tar, чтобы архивировать каталог в один файл .tgz, а затем каталог удалить. Будет весьма печально, если tar-файл по какой-то причине создать не удастся, а содержимое каталога все равно сотрется. Поэтому перед удалением нужно проверить, что tar завершилась успешно:

```
# Создадим несколько тестовых файлов
mkdir a_test_dir
echo testing ... testing > a_test_dir/tt
# Архивируем их и удалим, но только если архивирование прошло успешно
if tar cz a_test_dir > archived.tgz; then
   echo Compression went OK. Removing directory.
   rm -r a_test_dir
else
   echo Compression failed. Doing nothing.
fi
```

Если вы хотите посмотреть, как после первого прогона второй завершается неудачно, выполните команду chmod 000 archived.tgz, которая сделает запись в конечный архив невозможной, и запустите скрипт снова.

Не забывайте, что во всех показанных выше вариантах анализируется возвращаемое программой значение – с помощью test или иным способом. Но иногда требуется получить то, что программа выводит, и тут на помощь приходят обратные апострофы. Например, команда cat yourfile | wc -l порождает одно число – количество строк в файле yourfile (в предположении, что он существует), и это число можно проверить следующим образом:

```
if [ `cat yourfile | wc -l` -eq 0 ] ; then echo empty file.; fi
```

Настроим мультиплексор

У меня во время кодирования всегда открыты два терминала: в одном редактор с текстом программы, а в другом я компилирую и запускаю программу (возможно, в отладчике). А если исходных файлов несколько, то возникает настоятельная необходимость в удобном механизме перехода от одного терминала к другому.

Существуют два популярных терминальных мультиплексора по обе стороны от барьера, разделяющего вечных соперников, GNU и BSD: GNU Screen и tmux. Менеджер пакетов, скорее всего, сможет установить оба.

У обеих программ имеется единственная управляющая клавиша. В GNU Screen это по умолчанию **Ctrl-A**, а в tmux – **Ctrl-B**, но по общепринятому соглашению все переназначают клавишу на **Ctrl-A** с помощью добавления следующих команд:

```
unbind C-b
set -g prefix C-a
bind a send-prefix
```

в файл *.tmux_conf* в домашнем каталоге. В руководствах описаны еще десятки настроек в конфигурационных файлах. Кстати, когда будете искать советы и документацию, набирайте в поисковике *GNU Screen*, потому что по одному лишь слову *Screen* ничего полезного вы не получите.

Если в качестве управляющей клавиши задана **Ctrl+A**, то нажатием **Ctrl+A Ctrl+A** производится переход между двумя окнами, а в руководстве можно прочитать о прочих комбинациях **Ctrl+A** с другими клавишами, в частности о том, как перемещаться вперед и назад по списку окон и как вывести весь список окон, чтобы можно было выбрать из него нужное.

Таким образом, оба мультиплексора решают проблему нескольких окон. Но этим они отнюдь не ограничиваются.

- Комбинация **Ctrl+A D** отсоединяет сеанс, то есть на вашем терминале больше не отображаются различные виртуальные терминалы, управляемые мультиплексором. Но они продолжают работать в фоновом режиме.
  - В конце долгого рабочего дня в компании GNU Screen/Tmux отсоедините сеанс. А позже из дома или завтра утром присоедините его снова командой `screen -r` или `tmux attach`,и вы окажетесь точно там, где прервались. Возможность продолжать работу после отсоединения удобна и в случае неустойчивого подключения к серверу где-нибудь в Белизе или в Украине.
  - Мультиплексор не прерывает работу программ в своих виртуальных терминалах после отсоединения, что полезно, когда нужно запустить длительный процесс на ночь.
- Существует возможность копирования и вставки.
  - Находясь в режиме копирования, вы можете без использования мыши пролистать все, что недавно показывалось на экране терминала, выделить какой-то фрагмент, скопировать его во внутренний буфер обмена мультиплексора, а затем вставить скопированный текст в командную строку.
  - Просматривая текст в режиме копирования, вы можете перелистывать историю и искать строки.

Мультиплексоры – это последний шаг на пути превращения терминала из места работы в приятное место работы.

Команда fc

Команда fc (описанная в стандарте POSIX) превращает текст, набранный руками в оболочке, в допускающий повторное использование скрипт. Давайте попробуем.

```
fc -l # Флаг l означает "список", он важен
```

Теперь на экране появляется пронумерованный список нескольких последних команд. Возможно, в вашей оболочке того же эффекта можно достичь с помощью команды `history`.

Флаг `-n` подавляет вывод номеров строк, следующая команда выводит в файл элементы истории с номерами от 100 до 200:

```
fc -l -n 100 200 > a_script
```

После этого можно удалить из файла строки, знаменующие неудачные эксперименты, и тем самым превратить бестолковые игрища в командной строке в аккуратный скрипт оболочки.

Без флага -l команда fc становится более оперативным, но и менее постоянным инструментом. Она подгружает редактор (это означает, что перенаправление вывода с помощью > приведет к кажущемуся зависанию), отключает вывод номеров строк и при выходе из редактора немедленно выполняет все введенные в нем команды. Это очень удобно для быстрого повтора нескольких последних строк, но может привести к катастрофическим последствиям в случае небрежности. Если вы поняли, что забыли указать флаг -l, или недоумеваете, почему вдруг оказались в редакторе, немедленно удалите все находящееся на экране, чтобы случайно не выполнить то, что для выполнения не предназначено.

Но закончим все же в позитивном ключе: fc расшифровывается как *fix command* (исправить команду), и это ее самое простое применение. Без каких-либо флагов она позволяет редактировать только предшествующую строку; это удобно, когда нужно внести в команду нетривиальные исправления.

Пробуем другую оболочку

Помимо оболочки, которую поставщик вашей системы решил выбрать по умолчанию, в мире существует много других. Здесь я опишу кое-какие интересные вещи, которые умеет делать оболочка Z, – просто чтобы вы поняли, что может дать отказ от bash.

Перечни возможностей и переменных оболочки Z занимают несколько десятков страниц, экономии в ней не место – да и зачем по-спартански относиться к удобствам, которые несет с собой интерактивность? (Если вам больше по душе спартанская эстетика, но при этом вы все же хотите уйти от bash, попробуйте ash.) Задайте переменные в файле ~/.zshrc (или просто введите в командной строке для экспериментов); ниже показана переменная, которая понадобится в последующих примерах:

```
setopt INTERACTIVE_COMMENTS
# такие комментарии не приводят к ошибке
```

Расширение имен по маске, например замена file.* последовательностью file.c file.o file.h – обязанность любой оболочки. Но в zsh имеется полезное дополнение: **/ означает, что оболочка должна при расширении по маске рекурсивно обойти дерево каталогов. Согласно POSIX, вместо знака ~ подставляется домашний каталог, поэтому если вы хотите найти все *c*-файлы в своей области файловой системы, наберите ls ~/**/*.c.

Давайте сделаем резервные копии всех своих *c*-файлов:

```
# Эта строка может создать множество файлов в самых разных местах домашнего каталога
for ff in ~/**/*.c; do cp $ff ${ff}.bkup; done
```

Как вы, наверное, помните, в bash реализована только целочисленная арифметика, то есть $((3/2)) всегда равно 1. Оболочки Zsh и Ksh (и некоторые другие) похожи в этом отношении на C, то есть дают вещественный результат, если привести числитель или знаменатель к типу с плавающей точкой:

```
echo $((3/2.))  # работает в zsh, синтаксическая ошибка в bash

# Повторим приведенный ранее пример с подсчетом строк:
Files=`ls *.c |wc -l`
Lines=`grep '[)};]' *.c | wc -l`
```

```
# приводим к типу с плавающей точкой путем сложения с 0.0
echo lines/file = $(($Lines/($Files+0.0)))
```

Пробелы в именах файлов могут привести к ошибкам в bash, потому что пробелы разделяют элементы списка. В Zsh имеется синтаксис массива, не зависящий от использования пробела в качестве разделителя элементов.

```
# Создаем два файла, в имени одного из них есть пробелы
echo t1 > "test_file_1"
echo t2 > "test file 2"

# Дает ошибку в bash, работает в Zsh.
for f in test* ; do cat $f; done
```

Сменить оболочку можно двумя способами: с помощью команды chsh сделать изменение узаконенным на уровне входа в систему (меняется файл */etc/passwd*) или, если это почему-либо проблематично, добавить команду exec -l /usr/bin/zsh (вместо zsh можете указать любую другую оболочку) в последней строке файла *.bashrc*, в результате чего bash будет заменять себя указанной оболочкой при каждом запуске.
Если вы хотите, чтобы в makefile использовалась нестандартная оболочка, то включите в него строку:

```
SHELL=command -v zsh
```

(или укажите другую оболочку). Описанная в POSIX команда command -v выводит полный путь к указанной команде, так что помнить его необязательно. SHELL – необычная переменная в том смысле, что она должна быть задана либо в самом makefile, либо в виде аргумента make, поскольку переменную окружения с именем SHELL make игнорирует.

Файлы makefile и скрипты оболочки

Наверное, в каждом проекте существует уйма мелких процедур (подсчет слов, проверка правописания, прогон тестов, сохранение в системе управления версиями, выгрузка из системы управления версиями на удаленный сервер, резервное копирование), которые можно было бы автоматизировать с помощью скрипта оболочки. Но вместо того чтобы заводить по файлу из одной-двух строк на каждое задание, все это можно поместить в один makefile.

Файлы makefile обсуждались в разделе «Работа с файлами makefile» выше, но теперь, больше узнав об оболочке, мы поговорим о том, что еще может находиться в этих файлах. Ниже приведен пример из моей практики, в котором используются конструкция оболочки if/then и команда test. Сам я работаю с Git, но мне приходится иметь дело с тремя репозиториями Subversion, а названия и порядок выполнения команд Subversion я постоянно забываю. Зато их помнит makefile.

Пример 3.4 ❖ Использование if/then и test в makefile (make_bit)

```
push:
	@if [ "x$(MSG)" = 'x' ] ; then \         ❶
		echo "Usage: MSG='your message here.' make push"; fi
	@test "x$(MSG)" != 'x'                   ❷
	git commit -a -m "$(MSG)"
	git svn fetch
	git svn rebase
```

```
   git svn dcommit

pull:
   git svn fetch
   git svn rebase
```

❶ Каждая операция фиксации должна сопровождаться сообщением, которое передается в переменной окружения, задаваемой в командной строке: `MSG="This is a commit." make push`. Эта строка в предложении `if-then` печатает напоминание, если я забуду задать сообщение.

❷ Проверяю, что результат подстановки в строку `"x$(MSG)"` содержит что-нибудь, кроме `"x"`, то есть строка `$(MSG)` не пуста. Добавление `x` в строке по обе стороны знака равенства предотвращает ошибку в случае, когда `$(MSG)` пуста. Если проверка не проходит, то `make` больше ничего не делает.

Команды, запускаемые из makefile, в одних отношениях практически совпадают с запускаемыми непосредственно из командной строки, а в других кардинально отличаются.

- Каждая строка работает независимо от всех остальных, в отдельной оболочке. Если вы напишете в `makefile` такие строки:

  ```
  clean:
    cd junkdir
    rm -f *  # Не включайте такую команду в чmakefile.
  ```

  то вас постигнет глубокое разочарование. Эти строки эквивалентны такому коду на С:

  ```
  system("cd junkdir");
  system("rm -f *");
  ```

  Или поскольку `system("cmd")` эквивалентно `sh -c "cmd"`, то наш код в makefile эквивалентен такому:

  ```
  sh -c "cd junkdir"
  sh -c "rm -f *"
  ```

  Знатоки оболочки знают, что конструкция `(cmd)` запускает `cmd` в подоболочке, поэтому рассматриваемый фрагмент эквивалентен таким командам, введенным непосредственно в командной строке:

  ```
  (cd junkdir)
  (rm -f *)
  ```

  Во всех случаях вторая подоболочка ничего не знает о происходящем в первой. `make` сначала запускает новую оболочку, та переходит в каталог, который вы собираетесь очистить, после чего завершается. Затем `make` запускает еще одну оболочку, причем текущим является тот каталог, из которого была запущена сама команда `make`, и в ней вызывает `rm -f *`.
  Есть в этой ситуации и плюс – `make` удалит неправильно написанный makefile. А чтобы выразить первоначальное намерение, нужно написать:

```
cd junkdir && rm -f *
```

где оператор && осуществляет такое же короткое замыкание, как в C (то есть если первая команда завершается с ошибкой, то вторая не запускается). Или можно воспользоваться обратной косой чертой, чтобы соединить две строки в одну:

```
cd junkdir&& \
rm -f *
```

Впрочем, я в подобных случаях не стал бы доверять обратной косой черте. На практике гораздо надежнее написать `rm -f junkdir/*`.

- `make` заменяет вхождения `$x` (односимвольные имена переменных) или `$(xx)` (двухбуквенные имена переменных) значениями соответствующих переменных.
- Если вы хотите, чтобы подстановку выполняла оболочка, а не `make`, опустите скобки и каждый знак `$` замените двумя (`$$`). Например, чтобы с помощью оболочки создать резервные копии файлов из makefile, нужно написать: `for i in *.c; do cp $$i $${i%%.c}.bkup; done`.
- Вспомните рассмотренный выше прием – задание переменной окружения непосредственно перед командой, например `CFLAGS=-O3 gcc test.c`. Это может оказаться полезным в ситуации, когда каждая оболочка существует только на протяжении работы одной команды. Не забывайте, что присваивание должно располагаться прямо перед командой, а не перед ключевым словом оболочки типа `if` или `while`.
- Знак `@` в начале строки означает, что команду нужно выполнить, но печатаемые ей сообщения на экран не выводить.
- Знак `-` в начале строки означает, что даже если данная команда возвращает ненулевое значение, выполнение все равно нужно продолжать. По умолчанию работа makefile останавливается после первой же команды, вернувшей ненулевое значение.

Для простых проектов и большинства повседневных проблем применение возможностей оболочки в makefile может дать очень много. Вы знаете причуды своего компьютера, и makefile позволяет учесть их в одном месте и больше о них не думать.

Будет ли ваш makefile полезен коллегам? Если ваша программа представляет собой обычный набор *c*-файлов, если все необходимые библиотеки уже установлены и если переменные `CFLAGS` и `LDLIBS` в файле makefile соответствуют системе коллеги, то, быть может, все заработает нормально, а в худшем случае потребуется раз-другой связаться по электронной почте и уточнить детали. Если же вы генерируете разделяемую библиотеку, то даже не думайте передавать свой makefile – процедуры создания такой библиотеки на платформах Mac, Linux, Windows, Solaris и даже на разных версиях одной платформы очень сильно различаются. Предлагая свое творение широкой публике, нужно все автоматизировать по максимуму, потому что переписываться по поводу флагов с десятками или сотнями людей тяжеловато, да и большинство потенциальных пользователей просто не захочет

тратить столько времени и сил на то, чтобы заставить работать код, написанный каким-то незнакомцем. Поэтому для публично распространяемых пакетов необходим еще один слой программного обеспечения.

Создание пакета с помощью Autotools

Autotools – это как раз то, что позволяет вам скачать библиотеку или программу в исходных кодах и выполнить такие команды:

```
./configure
make
sudo make install
```

(и ничего больше) для ее сборки и установки. Только подумайте, до каких чудес дошла современная наука: разработчик понятия не имеет, какой у вас компьютер, где находятся ваши программы и библиотеки (в */usr/bin*? в */sw*? в */cygdrive/c/bin*?) и какие еще причуды характерны для вашей машины, и тем не менее скрипт `configure` во всем разобрался и сгенерировал работающий makefile. Таким образом, Autotools – на сегодня основной способ распространения кода. Если вы хотите, чтобы человек, незнакомый с вашим кодом, пользовался им (или чтобы ваша программа была включена в репозиторий пакетов какого-то дистрибутива Linux), то использование Autotools для сборки пакета сильно повышает ваши шансы на успех.

В сети полно пакетов, которым для установки нужен какой-то определенный каркас, например Scheme, Python версии ≥2.4, но <3.0, менеджер пакетов Red Hat Package Manager (RPM) и т. д. Каркас облегчает пользователям задачу установки пакета, только сначала нужно установить сам каркас. Для пользователей, не имеющих привилегий администратора, такое требование может оказаться чрезмерным. Autotools выделяется тем, что довольствуется совместимостью компьютера пользователя со стандартом POSIX.

Владение Autotools в совершенстве требует изрядных навыков, но основы просты. К концу этого раздела мы напишем скрипт сборки из шести строк, выполним четыре команды и получим полный (хотя и примитивный) пакет, готовый к распространению.

История развития Autoconf, Automake и Libtool довольно запутанна: это разные пакеты, и, в принципе, можно использовать их поодиночке. Но вот как я представляю себе эволюцию.

Менон. Люблю я `make`. Ах, как это прелестно – собрать в одном месте все мелкие шажки, необходимые для сборки проекта.

Сократ. Да, автоматизация – великое дело. Автоматизировать нужно все. И всегда.

Менон. В моем makefile куча целей, пользователь может набрать просто `make`, чтобы создать программу, или `make install` – чтобы установить ее, или `make check` – чтобы прогнать тесты, да мало ли... Написать все эти цели было трудненько, зато пользоваться – одно удовольствие.

Сократ. Так, а напишу-ка я систему – назову ее Automake, которая будет автоматически генерировать файлы makefile со всеми обычными целями из очень коротенького пра-makefile.

Менон. Отличная мысль. Особенно противно строить разделяемые библиотеки, потому что в каждой системе это делается по-своему.

Сократ. Воистину противно. Имея информацию о системе, я смогу написать программу генерации скриптов, потребных для создания разделяемых библиотек из исходного кода, а затем включить их в файлы makefile, порожденные Automake.

Менон. Здорово! Значит, мне остается только сказать тебе, какая у меня операционная система и как называется мой компилятор – cc, clang, gcc, или как-то еще – и ты включишь код, соответствующий моей системе.

Сократ. Э-э, так и до ошибок недалеко. А я вот напишу систему, скажем Autoconf, которая будет знать обо всех существующих системах и генерировать отчет, содержащий все, что Automake и твоя программа должны знать о системе. Тогда Autoconf сможет запустить Automake, который воспользуется сведениями в моем отчете, чтобы создать makefile.

Менон. Я просто поражен – ты автоматизировал процесс автогенерации makefile. Получается, что теперь мне нужно не изучать особенности различных платформ, а писать конфигурационные файлы для Autoconf и шаблоны makefile для Automake.

Сократ. Ты прав. А я должен буду написать инструмент, имя ему Autoscan, который просмотрит файл *Makefile.am*, подготовленный тобой для Automake, и автоматически сгенерирует файл *configure.ac* для Autoconf.

Менон. И, стало быть, останется только автоматически сгенерировать *Makefile.am*.

Сократ. Вот-вот. Читай документацию и сделай это сам.

На каждом шаге этой истории добавляется еще немножко автоматизации к достигнутому на предыдущем шаге. Automake с помощью простого скрипта генерирует файлы makefile (которые и так уже являются большим шагом вперед в деле автоматизации компиляции, по сравнению с набором команд вручную); Autoconf проверяет свое окружение и использует собранную информацию для запуска Automake; Autoscan просматривает ваш код и сообщает, что необходимо для успешной работы Autoconf. Libtool остается в тени и незаметно помогает Automake.

Пример работы с Autotools

В примере 3.5 приведен скрипт, с помощью которого Autotools сделает все необходимое для программы *Hello, World*. Это скрипт оболочки, который вы можете скопировать в командную строку (позаботьтесь только, чтобы после знаков обратной косой черты не было пробелов). Разумеется, прежде всего нужно с помощью менеджера пакетов установить все части Autotools: Autoconf, Automake и Libtool.

Пример 3.5 ❖ Сборка пакета программы Hello, World (auto.conf)

```
if [ -e autodemo ]; then rm -r autodemo; fi
mkdir -p autodemo                              ❶
cd autodemo
cat > hello.c <<\
"--------------"
#include <stdio.h>

int main(){ printf("Hi.\n"); }
--------------

cat > Makefile.am <<\                          ❷
"--------------"
bin_PROGRAMS=hello
hello_SOURCES=hello.c
--------------

Autoscan                                       ❸
sed -e 's/FULL-PACKAGE-NAME/hello/' \
    -e 's/VERSION/1/' \                        ❹
    -e 's|BUG-REPORT-ADDRESS|/dev/null|' \
    -e '10i\
AM_INIT_AUTOMAKE' \
        < configure.scan > configure.ac

touch NEWS README AUTHORS ChangeLog            ❺

autoreconf -iv                                 ❻
./configure
make distcheck
```

❶ Создать каталог и с помощью встроенного документа записать в него файл *hello.c*.

❷ Нам понадобится написанный вручную файл *Makefile.am*, состоящий всего из двух строк. Даже строка `hello_SOURCES` необязательна, потому что Automake может догадаться, что программа *hello* строится из исходного файла *hello.c*.

❸ `autoscan` генерирует файл *configure.scan*.

❹ Отредактировать файл *configure.scan* – включить в спецификацию параметры вашего проекта (название, версию, контактный адрес электронной почты) и добавить строку `AM_INIT_AUTOMAKE` для инициализации Automake. (Да, это странно, особенно принимая во внимание, что Autoscan пользовался относящимся к Automake файлом *Makefile.am* для сбора информации, поэтому он прекрасно знает, что мы собираемся работать с Automake.) Можно было бы сделать это вручную; я воспользовался потоковым редактором sed, чтобы вписать нужный мне номер версии прямо в *configure.ac*.

❺ Наличия этих четырех файлов требует стандарт кодирования GNU, а потому GNU просто отказывается работать без них. Я смошенничал – создал пустые файлы с помощью стандартной команды `touch`; в ваших файлах должна присутствовать реальная информация.

❻ Имея файл *configure.ac*, запускаем `autoreconf`, чтобы сгенерировать все файлы, нужные для распространения (и в первую очередь `configure`). При наличии флага `-i` создаются дополнительные стереотипные файлы, необходимые системе.

Что создают все эти макросы? Сам файл *hello.c* занимает всего три строки, *Makefile.am* и того меньше – две строки, так что руками пользователя написано пять строчек. Что касается остального, то команда `wc -l *`, запущенная в каталоге после завершения работы скрипты, насчитала примерно 11 000 строк текста, из которых 4700 пришлось на скрипт `configure`. На вашей машине результаты могут немного отличаться.

Такой немаленький размер объясняется переносимостью: у получателей пакета может не оказаться Autotools, и одному Богу известно, чего еще у них нет, поэтому созданный скрипт может полагаться только на средства, требуемые стандартом POSIX.

В файле makefile из 600 строк я насчитал 73 цели.

- Цель по умолчанию, которая строится, когда введена только команда `make` без параметров, она порождает исполняемый файл.
- Команда `sudo make install` установит программу, если вы захотите. Команда `sudo make uninstall` удалит ее.
- Существует даже взрывающая мозг цель `make Makefile` (она бывает полезна, если вы подправите файл *Makefile.am* и захотите быстро перегенерировать makefile).
- Автору пакета интересна цель `make distcheck`, которая генерирует tar-файл, содержащий все необходимое пользователю для распаковки и запуска обычной последовательности команд `./configure && make && sudo make install` (без помощи со стороны системы Autotools, которая имеется на машине, где вы ведете разработку) и проверяет правильность собранного дистрибутива, в том числе прогоняет заданные вами тесты.

На рис. 3.1 вся эта история представлена в виде блок-схемы.

Вам предстоит написать только два из показанных файлов (в закрашенных блоках), все остальное генерируется автоматически соответствующей командой. Начнем с нижней части: пользователь получает ваш пакет в виде tgz-файла и распаковывает его командой `tar xvzf your_pkg.tgz`, которая создает каталог, содержащий ваш код, файлы *Makefile.am*, *configure* и еще ряд вспомогательных файлов, которые мы здесь обсуждать не станем. Пользователь вводит команду `./configure`, она генерирует файлы *configure.h* и *Makefile*. Теперь все готово к вводу команд `make; sudo make install`.

Задача автора – сгенерировать tgz-файл, содержащий высококачественные файлы *configure* и *Makefile.am*, при запуске которых у пользователя не возникнет никаких проблем. Для начала напишите сами файл *Makefile.am*. Выполните `autoscan` и созданный предварительный файл *configure.scan* отредактируйте вручную, чтобы получить *configure.ac*. (На схеме не показаны четыре файла, требуемых согласно стандартам кодирования GNU: *NEWS*, *README*, *AUTHORS* и *ChangeLog*).

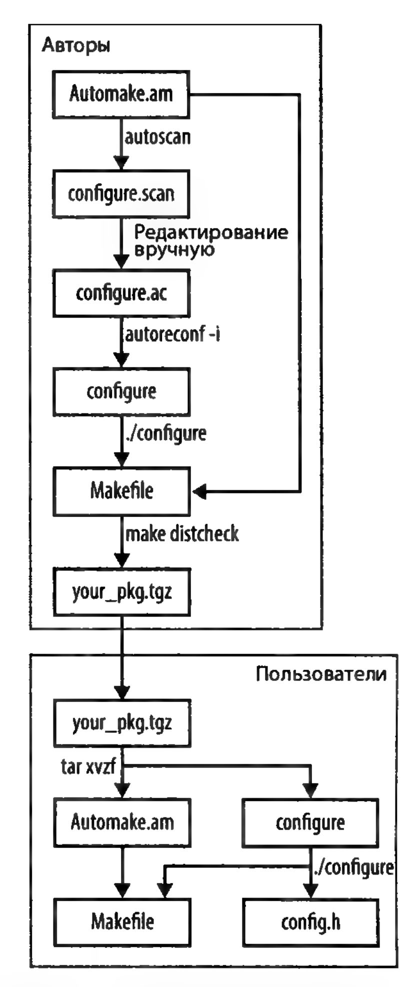


Рис. 3.1 ❖ Блок-схема Autotools.
Вам нужно написать только два из показанных файлов (в закрашенных блоках), все остальное генерируется автоматически соответствующей командой

Затем выполните команду `autoreconf -iv`, которая создаст скрипт `configure` (и много других вспомогательных файлов). Этот скрипт можно теперь запустить и получить makefile, а имея makefile, выполнить `make distcheck` для генерации дистрибутивного tgz-файла.

Обратите внимание на некоторое перекрытие: вы используете те же файлы *configure* и *Makefile*, что и пользователь, только ваша цель – собрать пакет, а цель пользователя – этот пакет установить. Это означает, что у вас имеются средства установить и протестировать код, не собирая полного пакета, а у пользователя – возможность пересобрать пакет по-другому, если возникнет такое желание.

Описание Makefile с помощью Makefile.am

Типичный makefile наполовину описывает структуру зависимостей частей вашего проекта друг от друга, а на другую половину содержит разного рода переменные и исполняемые процедуры. Ваш файл *Makefile.am* должна в первую очередь интересовать структура – что подлежит компиляции и что от чего зависит, а конкретные детали подставят Autoconf и Automake, руководствуясь встроенными в них знаниями о компиляции на различных платформах.

В *Makefile.am* есть два типа информации, которые я буду называть *переменными формы* и *переменными содержания*.

Переменные формы

У файлов, которые должны быть обработаны с помощью makefile, может быть различное назначение, и в Automake для каждого назначения предусмотрена аннотация в виде короткой строки.

bin
Установить туда, где находятся другие программы, например в */usr/bin* или в */usr/local/bin.*

include
Установить туда, где находятся заголовки, например в */usr/local/include.*

lib
Установить туда, где находятся библиотеки, например в */usr/local/lib.*

pkgbin
Если проект именованный, установить в подкаталог каталога с главной программой, например в */usr/local/bin/project/* (и аналогично для *pkginclude* и *pkglib*).

check
Используется для тестирования программы, когда пользователь вводит команду `make check`.

noinst
Не устанавливать, просто оставить файл, он будет нужен для обработки другой цели.

Automake генерирует стереотипные скрипты для `make` и имеет различные шаблоны для следующих разделов:

`PROGRAMS`	
`HEADERS`	
`LIBRARIES`	*статические библиотеки*
`LTLIBRARIES`	*разделяемые библиотеки, сгенерированные Libtool*
`DIST`	*файлы, распространяемые вместе с пакетом, например файлы данных, которые ни в какую другую категорию не попадают*

Назначение плюс шаблонный формат – это и есть переменная формы, например:

bin_PROGRAMS	*программы, которые нужно собрать и установить*
check_PROGRAMS	*программы, которые нужно собрать для тестирования*
include_HEADERS	*заголовки, которые нужно установить в системный каталог include*
lib_LTLIBRARIES	*динамические и разделяемые библиотеки, собираемые с помощью Libtool*
noinst_LIBRARIES	*статические библиотеки (собираемые без участия Libtool), которые могут понадобиться позднее*
noinst_DIST	*распространять с пакетом, но и только*
python_PYTHON	*написанный на Python код, который нужно откомпилировать в байт-код и установить туда, куда устанавливаются пакеты на Python*

Определившись с формой, мы можем воспользоваться этими переменными, чтобы указать, как должен обрабатываться каждый файл. В нашем примере пакета "Hello, World" был всего один файл:

```
bin_PROGRAMS = hello
```

В качестве другого примера рассмотрим раздел `noinst_DIST`, в котором описываются данные, необходимые для тестов, прогоняемых после компиляции, – устанавливать их никуда не нужно. В каждой строке может быть перечислено произвольное количество файлов. Например:

```
pkginclude_HEADERS = firstpart.h secondpart.h
noinst_DIST = sample1.csv sample2.csv \
            sample3.csv sample4.csv
```

Переменные содержания

Элементы из раздела `noinst_DIST` просто копируются в дистрибутивный пакет, а из раздела `HEADERS` – копируются в конечный каталог с установкой соответствующих разрешений. С ними, стало быть, все ясно.

Для выполнения шагов компиляции, например `..._PROGRAMS` и `..._LDLIBRARIES`, Automake должен знать многочисленные детали. Как минимум необходимо знать, какие исходные файлы компилировать. Следовательно, для каждого элемента справа от знака равенства в строке переменной формы, относящейся к компиляции, нужна переменная, определяющая исходные файлы. Например, для компиляции следующих двух программ нужны две строки `SOURCES`:

```
bin_PROGRAMS= weather wxpredict
weather_SOURCES= temp.c barometer.c
wxpredict_SOURCES=rng.c tarotdeck.c
```

Для простого пакета этим все может и исчерпываться.



Здесь мы имеем еще одно нарушение принципа, согласно которому вещи, предназначенные для разных целей, должны и выглядеть по-разному: переменные содержания записываются в таком же виде `lower_UPPER`, как и переменные формы, но образуются из совершенно других частей и служат совершенно другим целям.

Напомним: обсуждая старые добрые файлы makefile, мы упомянули, что в `make` встроен ряд правил по умолчанию, в которых переменные типа `CFLAGS` используются для настройки деталей операций. Переменные формы Automake по существу определяют дополнительные правила по умолчанию, и с каждым из них ассоциирован свой набор переменных.

Например, правило компоновки объектных файлов в исполняемые может выглядеть следующим образом:

```
$(CC) $(LDFLAGS) temp.o barometer.o $(LDADD) -o weather
```



В программе GNU Make во второй части команды компоновки используется переменная LDLIBS, а в GNU Automake – переменная LDADD.

С помощью поисковика нетрудно найти в Интернете документацию, в которой объясняется, как из той или иной переменной формы получается набор целей в итоговом makefile, но мне кажется, что самый простой способ выяснить, что делает Automake, – просто запустить программу и изучить сгенерированный makefile в редакторе.

Все эти переменные можно установить на уровне одной программы или библиотеки, например weather_CFLAGS=-O1. Либо воспользоваться переменной AM_VARIABLE, чтобы установить одно и то же значение для всех операций компиляции или компоновки. Вот как можно задать мои любимые флаги компилятора, описанные в разделе «Работа с файлами makefile» выше:

```
AM_CFLAGS=-g -Wall -O3
```

Я не включил флаг -std=gnu99, заставляющий gcc придерживаться не столь устаревшего стандарта, поскольку этот флаг зависит от компилятора. Если бы я добавил в файл *configure.ac* макрос AC_PROG_CC_C99, то Autoconf присвоил бы переменной CC значение gcc -std=gnu99. Autoscan (пока) недостаточно сообразителен, чтобы самостоятельно включить эту директиву в генерируемый файл *configure.scan*, так что ее, вероятно, придется добавить вам. (На момент написания этой книги еще не существовало макроса AC_PROG_CC_C11[1].)

Более специфичные правила переопределяют правила, основанные на макросах вида AM_, поэтому если вы хотите придерживаться общих правил, но переопределить какой-нибудь флаг в конкретном случае, то нужно поступить следующим образом:

```
AM_CFLAGS=-g -Wall -O3
hello_CFLAGS = $(AM_CFLAGS) -O0
```

Добавление средств тестирования

Я еще не рассказал о библиотеке для работы со словарями (она рассматривается в разделе «Расширение структур и словарей» на стр. 251), но продемонстрировал тестовую оснастку для нее в разделе «Автономное тестирование» выше. После того как Autotools соберет библиотеку, тесты имеет смысл прогнать еще раз. Таким образом, нам предстоит собрать:

- Библиотеку, включающую откомпилированные файлы *dict.c* и *keyval.c*. С ней ассоциированы заголовки *dict.h* и *keyval.h*, которые нужно распространять вместе с библиотекой.

[1] В 2012 году этот макрос был добавлен. – *Прим. перев.*

- Тестовую программу, причем Automake должен знать, что она предназначена именно для тестирования и устанавливать ее не нужно.
- Программу dict_use, которая пользуется библиотекой.

Эта повестка воплощена в жизнь в примере 3.6. Первой собирается библиотека, чтобы ее можно было использовать при сборке программы и тестовой оснастки. Переменная TESTS определяет, какие программы или скрипты должны запускаться, когда пользователь вводит команду make check.

Пример 3.6 ❖ Файл Automake, в котором учтено тестирование (dict.automake)

```
AM_CFLAGS=`pkg-config --cflags glib-2.0` -g -O3 -Wall   ❶

lib_LTLIBRARIES=libdict.la                               ❷
libdict_la_SOURCES=dict.c keyval.c                       ❸

include_HEADERS=keyval.h dict.h

bin_PROGRAMS=dict_use
dict_use_SOURCES=dict_use.c
dict_use_LDADD=libdict.la                                ❹

TESTS=$(check_PROGRAMS)                                  ❺
check_PROGRAMS=dict_test
dict_test_LDADD=libdict.la
```

❶ Я немного смошенничал, потому что у других пользователей программы pkg-config может не оказаться. Без предположения о наличии pkg-config лучшее, что можно сделать, – проверить присутствие библиотеки с помощью макросов Autoconf AC_CHECK_HEADER и AC_CHECK_LIB, и если она не найдена, то попросить пользователя изменить переменные окружения CFLAGS или LDFLAGS, указав в них правильные флаги -I или -L. Поскольку мы еще не дошли до обсуждения *configure.ac*, я просто воспользуюсь pkg-config.

❷ Первым делом мы должны собрать разделяемую библиотеку (с помощью Libtool, отсюда и префикс LT в имени LTLIBRARIES).

❸ Чтобы образовать переменную содержания из имени файла, замените все символы, отличные от букв, цифр и знака @, символом подчеркивания, как в случае libdict.la ® libdict_la.

❹ Определив, как генерировать разделяемую библиотеку, мы можем использовать ее для сборки программ и тестов.

❺ Переменная TESTS определяет, какие тесты прогонять в ответ на команду make check. Поскольку зачастую это скрипты оболочки, не требующие компиляции, эта переменная отличается от переменной check_PROGRAMS, которая определяет, какие программы необходимо собрать для выполнения тестирования. В нашем случае они совпадают, поэтому мы присваиваем одну другой.

Добавление фрагментов makefile

Если вы провели исследовательскую работу и выяснили, что Automake не может справиться с некоторой специфической целью, то можете вписать ее в *Makefile.am*, как в обычный makefile, – просто включите цель и связанные с ней действия:

```
target: deps
    script
```

в любое место *Makefile.am*, и Automake буквально скопирует ее в конечный makefile. Например, в файле *Makefile.am*, описанном в разделе «Python как включающий язык» на стр. 128, явно указано, как компилировать пакет на Python, потому что Automake этого не знает (он умеет лишь компилировать в байт-код одиночные *py*-файлы).

Переменные, не удовлетворяющие форматам Automake, также копируются буквально. Это особенно полезно в сочетании с Autoconf, потому что если в *Makefile.am* имеются такие присваивания переменным:

```
TEMP=@autotemp@
HUMIDITY=@autohum@
```

а в файле *configure.ac* есть такой код:

```
# configure - обычный скрипт оболочки, а это обычные переменные оболочки
autotemp=40
autohum=.8
AC_SUBST(autotemp)
AC_SUBST(autohum)
```

то в конечном makefile появится текст вида:

```
TEMP=40
HUMIDITY=.8
```

Таким образом, мы получаем прямой канал из скрипта оболочки, формируемого Autoconf, в конечный makefile.

Скрипт configure

Скрипт оболочки *configure.ac* порождает два продукта: файл makefile (с помощью Automake) и файл-заголовок *config.h*.

Если вы открывали какой-нибудь из сгенерированных к этому моменту файлов *configure.ac*, то, наверное, обратили внимание, что код в нем совсем не похож на скрипт оболочки. Дело в том, что в нем используются многочисленные макросы (написанные на макроязыке m4), предопределенные в Autoconf. Но будьте уверены – каждый такой макрос расширяется в строки на привычном языке оболочки. Таким образом, *configure.ac* – не рецепт и не спецификация, по которой порождается скрипт оболочки *configure*, это и есть *configure*, только сжатый с помощью весьма выразительных макросов.

Синтаксис языка m4 несложен. Каждый макрос внешне напоминает функцию – после имени макроса в скобках записывается список аргументов, разделенных запятыми (если аргументов нет, то скобки обычно опускают). Там, где в большинстве языков мы написали бы `'литеральный текст'`, в m4 пишут `[литеральный текст]`, и чтобы избежать сюрпризов из-за того, что m4 слишком агрессивно разбирает входные данные, заключайте все параметры макросов в квадратные скобки.

Хорошим примером может служить первая же строка, генерируемая Autoscan:

```
AC_INIT([FULL-PACKAGE-NAME], [VERSION], [BUG-REPORT-ADDRESS])
```

Мы знаем, что этот макрос сгенерирует несколько сотен строк на языке оболочки и где-то среди них будут установлены и интересующие нас элементы. Вместо строк в квадратных скобках подставьте нужные значения. Часто некоторые элементы опускаются. Например, можно написать

```
AC_INIT([hello], [1.0])
```

если вы не хотите получать извещения об ошибках от пользователей. Можно даже совсем не передавать аргументов, как в случае макроса AC_OUTPUT, и тогда ставить скобки необязательно.



В текущей версии документации по m4 принято обозначать необязательные аргументы – да-да, я не шучу – квадратными скобками. Поэтому имейте в виду, что в написанных на m4 макросах для Autoconf квадратные скобки означают литеральный, то есть нерасширяемый, текст, а в документации по m4 – необязательный аргумент.

Какие макросы нужны в реальном файле Autoconf? Перечислим их в том порядке, в котором они встречаются.

- AC_INIT(...), показан выше.
- AM_INIT_AUTOMAKE, чтобы Automake сгенерировал makefile.
- LT_INIT настраивает Libtool, это нужно, только если вы собираетесь устанавливать разделяемую библиотеку.
- AC_CONFIG_FILES([Makefile subdir/Makefile]) инструктирует Autoconf обработать перечисленные файлы и заменить переменные вида @cc@ соответствующими значениями. Если предполагается создать несколько файлов makefile (обычно в подкаталогах), их нужно перечислить здесь.
- AC_OUTPUT, чтобы вывести результат.

Итак, у нас имеется спецификация для создания пакета сборки, работающего в любой POSIX-совместимой системе, и занимает она четыре или пять строк, три из которых, вероятно, написал за вас Autoscan.

Но настоящее искусство, требующееся для того, чтобы получить не просто работающий, а интеллектуальный *configure.ac*, – научиться предвидеть проблемы, с которыми может столкнуться пользователь, и отыскать макрос Autoconf, который распознает проблему и, возможно, исправит ее. Один такой пример мы уже видели: я рекомендовал включить в *configure.ac* макрос AC_PROG_CC_C99, который проверит наличие компилятора, отвечающего стандарту C99. Стандарт POSIX требует, чтобы такой компилятор присутствовал и назывался c99, но из того, что POSIX что-то требует, еще не следует, что все системы подчиняются, и это как раз тот аспект, который должен проверять хороший скрипт *configure*.

Наличие необходимых библиотек – классический пример проверки предварительных условий. Вернемся на секунду к файлам, которые генерирует Autoconf; *config.h* – это обычный С-заголовок, состоящий из последовательности директив #define. Например, если Autoconf убедился в наличии библиотеки GSL, то в этом файле появится такая строка:

```
#define HAVE_LIBGSL 1
```

А в своем коде вы можете расставить директивы #ifdef, так чтобы программа правильно вела себя вне зависимости от того, есть эта библиотека или нет.

Логика проверки, встроенная в Autoconf, не просто находит библиотеку с определенным именем в надежде, что она будет работать. Autoconf генерирует программу, которая не делает ничего полезного, а только вызывает какую-нибудь функцию из библиотеки, а затем пытается скомпоновать эту программу с библиотекой. Если компоновка завершается успешно, то есть уверенность, что компоновщик сможет найти и использовать библиотеку. Но Autoscan не может сам сгенерировать такую проверку, потому что не знает, какие функции в библиотеке имеются. Поэтому макросу проверки передаются имя библиотеки и имя функции, например:

```
AC_CHECK_LIB([glib-2.0],[g_free])
AC_CHECK_LIB([gsl],[gsl_blas_dgemm])
```

Добавьте в *configure.ac* по одной строке для каждой библиотеки, которую собираетесь использовать и наличие которой не гарантируется на 100% стандартом С. Из этих однострочных семян вырастут пышные плоды – фрагменты кода в скрипте configure.

Вспомните, что менеджеры пакетов разбивают библиотеки на два пакета: двоичный разделяемый объект (so-файл) и пакет для разработчиков, содержащий заголовки. Пользователи библиотеки могут забыть об установке пакета с заголовками (и даже не знать о нем), поэтому проверяйте его наличие, например вот так:

```
AC_CHECK_HEADER([gsl/gsl_matrix.h], , [AC_MSG_ERROR(
   [Не найдены файлы-заголовки GSL (производился поиск \
   <gsl/gsl_matrix.h> в пути для включаемых файлов). Если вы пользуетесь \
   менеджером пакетов, не забудьте установить пакет \
   libgsl-devel помимо самого пакета libgsl.])])
```

Обратите внимание на две запятые: у этого макроса три аргумента, (проверяемый заголовок, действие, когда найден, и действие, когда не найден), но второй из них мы не задаем.

Что еще могло бы случиться во время компиляции? Трудно стать авторитетом по всем причудам всех компьютеров на свете, имея в своем распоряжении всего одну-две машины. Autoscan дает ряд хороших советов, и не исключено, что при запуске autoreconf будут выданы дополнительные предупреждения по поводу того, что добавить в *configure.ac*. Будет правильно последовать рекомендациям. Но, на мой взгляд, самый лучший справочный материал – длинный перечень подлинных выдержек из стандарта POSIX, ошибок реализации и практических советов – это само руководство по Autoconf. В нем перечислены дефекты, которые Autoconf исправляет сам[1] и которые, следовательно (и слава Богу), для нас не представляют

[1] Например, «Оболочка dtksh в Solaris 10 и POSIX-оболочка в UnixWare 7.1.1 ... неправильно расширяют заключенные в скобки переменную, если она пересекает границу буфера размером 1024 или 4096 байтов во встроенном документе».

интереса, имеются рекомендации относительно написания кода, а описания некоторых системных капризов сопровождаются именем макроса Autoconf, который следует включить в файл *configure.ac* проекта, если ситуация того требует.

Дополнительный код на языке оболочки

Поскольку *configure.ac* – сжатый вариант скрипта *configure*, запускаемого пользователем, вы можете включить в него произвольный код на языке оболочки. Но прежде чем делать это, тщательно проверьте, нет ли уже макросов, которые делают то, что вам нужно, – так ли уникальна ваша ситуация, что раньше никто из пользователей Autotools с ней не сталкивался?

Если нужного макроса нет в самом пакете Autoconf, поищите в архиве макросов GNU Autoconf. Найденные макросы можно сохранить в подкаталоге *m4* каталога вашего проекта, и тогда Autoconf сможет их найти и использовать. См. также [Calcote 2010] – кладезь бесценной технической информации об Autotools.

Пользователю можно напечатать сообщение об успешном завершении процедуры конфигурирования, и макрос тут не нужен, потому что, кроме команды echo, ничего не используется. Вот пример такого сообщения:

```
echo \
"-------------------------------------------------------------

Thank you for installing ${PACKAGE_NAME} version ${PACKAGE_VERSION}.

Installation directory prefix: '${prefix}'.
Compilation command: '${CC} ${CFLAGS} ${CPPFLAGS}'

Now type 'make; sudo make install' to generate the program
and install it to your system.

-------------------------------------------------------------"
```

В этом сообщении встречаются переменные, определенные Autoconf. Все они описаны в документации, но можно также найти определенные переменные оболочки в самом скрипте *configure*.

Более развернутый пример использования Autotools, относящийся к библиотеке на Python, см. в разделе «Python как включающий язык» на стр. 128.

Глава 4

Управление версиями

Посмотри на мир через солнцезащитные очки,
Для рабочего класса жизнь покажется куда лучше.
— Gang of Four «I Found that Essence Rare»

Эта глава посвящена системам управления версиями (СУВ), в которых хранятся мгновенные снимки различных версий проекта, существовавших в ходе его разработки, например: этапов работы над этой книгой, вариантов написанного в муках любовного письма или какой-то программы. Система управления версиями дает нам ряд весьма важных возможностей.

- У файловой системы появляется новое измерение – время, так что мы можем узнать, как выглядел файл на прошлой неделе и что с тех пор изменилось. Даже если бы больше ничего не предлагалось, одна только эта возможность позволяет мне как автору работать более уверенно.
- Мы можем отслеживать несколько версий проекта, например мой экземпляр и экземпляр моего соавтора. И даже в пределах своего экземпляра я, возможно, захочу иметь версию проекта (*ветвь*) с экспериментальными функциями и хранить ее отдельно от стабильной версии, в которой не должно быть неприятных сюрпризов.
- На сайте **http://github.com** хранится примерно 175 000 проектов, большая часть которых, если судить по опубликованным в них же сведениям, написана на С. Существуют также серверы, где размещены репозитории СУВ поменьше, например GNU Savannah. Даже если вы не собираетесь модифицировать код, клонирование такого репозитория – быстрый способ скопировать программу или библиотеку на свой диск и использовать затем по собственному усмотрению. Когда проект готов к передаче в руки общественности (или до того), репозиторий можно сделать открытым и тем самым организовать еще один канал распространения.
- Когда два человека имеют доступ к одному и тому же проекту и оба могут вносить изменения в код, возникает проблема объединения модификаций – и система управления версиями позволяет сделать это с минимальными усилиями.

В этой главе рассматривается Git, *распределенная система управления версиями*; это означает, что любая копия проекта представляет собой автономный репозиторий со своей историей. Есть и другие системы в этой категории, например Mercurial и Bazaar. Между функциями всех таких систем существует более-менее взаимно однозначное соответствие, а имевшиеся ранее значительные отличия с

годами сгладились, так что, прочитав эту главу, вы сможете выбрать любую систему по своему вкусу.

Получение списка отличий с помощью diff

Самая рудиментарная форма управления версиями – использование утилит diff и patch, которые определены в стандарте POSIX и потому, скорее всего, установлены в вашей системе. Наверное, на вашем диске найдутся два похожих файла, а если нет, возьмите любой текстовый файл, измените в нем несколько строк и сохраните под другим именем. Затем выполните команду

```
diff f1.c f2.c
```

В ответ вы получите распечатку, рассчитанную скорее на машину, чем на человека, в которой перечислены отличающиеся строки. Если перенаправить вывод в текстовый файл командой `diff f1.c f2.c > diffs` и открыть файл diffs в редакторе, то вы получите раскрашенную картинку, в которой легко ориентироваться. Вы увидите строки с указанием имени и местоположения файла, возможно, несколько контекстных строк, совпадающих в обоих файлах, и строки, начинающиеся знаком + (добавленные) или – (удаленные). При запуске diff с флагом -u добавленные или удаленные строки будут окружены строками неизменившегося контекста.

Если есть две версии проекта v1 и v2 в разных каталогах, то, указав флаг -r (рекурсивно), можно создать один файл различий в унифицированном формате diff для всего каталога:

```
diff -ur v1 v2 > diff-v1v2
```

Команда patch читает файлы различий и вносит перечисленные в нем изменения. Если у вас и у коллеги имеется версия v1 проекта, то вы можете отправить ему файл diff-v1v2, а он выполнит команду

```
patch < diff-v1v2
```

и внесет все ваши изменения в свой экземпляр версии v1.

И даже если вы работаете в одиночестве, можете время от времени выполнять diff и в результате получать перечень произведенных за период изменений. Если в коде обнаружится ошибка, то файлы различий – первое, куда нужно смотреть, чтобы понять, что изменилось. Если вы уже удалили версию v1, то можете применить процедуру обратного латания к каталогу v2 – `patch -R < diff-v1v2` – и тем самым откатиться от версии 2 к версии 1. Если текущей является версия 4, то теоретически можно обработать несколько файлов различий, чтобы вернуться во времени еще раньше:

```
cd v4
patch -R < diff-v3v4
patch -R < diff-v2v3
patch -R < diff-v1v2
```

Я говорю *теоретически*, потому что хранить такую последовательность файлов различий утомительно и чревато ошибками. Система управления версиями сама позаботится о создании и хранении информации о различиях, или *дельтах*.

Объекты Git

Git – программа, написанная на C, и, как в любой подобной программе, в ней есть небольшой набор объектов. Основным является объект фиксации (commit), по смыслу близкий к унифицированному файлу различий. В каждом объекте фиксации инкапсулированы предыдущий объект фиксации и набор изменений, произведенных после его создания. Объект фиксации пользуется поддержкой со стороны *индекса* – списка изменений, зарегистрированных с момента создания последнего объекта фиксации; индекс нужен в первую очередь для генерации следующего объекта фиксации.

Объекты фиксации связываются друг с другом и образуют древовидную структуру. У каждого объекта фиксации есть по меньшей мере один родительский объект. Проход вверх и вниз по дереву сродни использованию команд `patch` и `patch -R` для версий.

Сам репозиторий формально не представлен каким-то одним объектом в исходном коде Git, но мне удобно рассматривать его как объект, потому что типичные операции – создание нового, копирование и освобождение – применяются ко всему репозиторию. Чтобы создать новый репозиторий в текущем рабочем каталоге, нужно выполнить команду:

```
git init
```

Теперь у вас имеется готовая к использованию система управления версиями. Возможно, вы ее не видите, потому что Git хранит все свои файлы в каталоге `.git`, а наличие точки означает, что стандартные утилиты, в частности `ls`, не показывают каталога. Но его можно увидеть с помощью команды `ls -a` или включив режим *показа скрытых файлов* в файловом менеджере.

Можно вместо этого клонировать репозиторий командой `git clone`. Именно таким образом мы получаем проект из репозитория Savannah или Github. Чтобы получить исходный код самой программы Git с помощью `git`, нужно написать:

```
git clone https://github.com/gitster/git.git
```

Возможно, вам будет интересно также клонировать репозиторий, содержащий примеры из этой книги:

```
git clone https://github.com/b-k/21st-Century-Examples.git
```

Если вы хотите изменить какой-то файл в репозитории `~/myrepo`, но опасаетесь что-нибудь сломать, перейдите во временный каталог (например, `mkdir ~/tmp; cd ~/tmp`), клонируйте свой репозиторий командой `git clone ~/myrepo` и экспериментируйте сколько душе угодно. Потом можете удалить клон (`rm -rf ~/tmp/myrepo`), на оригинале это никак не отразится.

Поскольку все сведения о репозитории хранятся в подкаталоге `.git` каталога вашего проекта, то для освобождения репозитория нужно всего лишь выполнить команду:

```
rm -rf .git
```

Такая высокая степень автономности репозитория означает, что можно без особых хлопот делать резервные копии и носить их с работы домой и обратно, копировать все во временный каталог для безопасного экспериментирования и т. п.

Мы почти готовы к созданию объектов фиксации, но поскольку в них инкапсулированы изменения, произошедшие с начального момента или с момента предыдущей фиксации, то необходимо что-нибудь изменить. Индекс (в исходном коде Git структура `struct index_state`) – это список изменений, которые будут включены в следующий объект фиксации. Индекс нужен для того, чтобы не регистрировать абсолютно все изменения в каталоге проекта. Например, файл `gnomes.o` и исполняемый файл `gnomes` порождаются из `gnomes.c` и `gnomes.h`. СУВ должна отслеживать изменения только в `gnomes.c` и `gnomes.h`, считая, что остальные файлы можно в любой момент сгенерировать заново. Таким образом, основная операция над индексом – добавление новых элементов в список изменений. Выполните команду:

```
git add gnomes.c gnomes.h
```

для добавления указанных файлов в индекс. Другие типичные операции изменения списка отслеживаемых файлов также должны быть зарегистрированы в индексе:

```
git add newfile
git rm oldfile
git mv flie file
```

Изменения, внесенные в файлы, которые уже отслеживаются Git, автоматически в индекс не добавляются, что, наверное, удивительно для пользователей других СУВ (см., однако, ниже). Добавьте их по отдельности для каждого файла командой `git add changedfile` или воспользуйтесь командой:

```
git add -u
```

чтобы добавить в индекс изменения во всех уже известных Git файлах. В какой-то момент накопится достаточно изменений, чтобы записать их в репозиторий в виде объекта фиксации. Новый объект фиксации создается командой:

```
git commit -a -m "Это первая фиксация."
```

Флаг `-m` присоединяет сообщение к новой ревизии, впоследствии оно будет выведено командой `git log`. Если сообщение отсутствует, то Git запустит текстовый редактор, указанный в переменной окружения `EDITOR`, чтобы его можно было ввести (по умолчанию обычно подразумевается редактор vi; если вам больше нравится другой редактор, экспортируйте эту переменную в скрипте инициализации оболочки, например `.bashrc` или `.zshrc`).

Флаг -a говорит Git, что я вполне мог по рассеянности не выполнить команду git add -u, поэтому хорошо бы сделать это перед фиксацией. На практике это означает, что запускать git add -u явно не нужно, коль скоро вы всегда задаете флаг -a в команде git commit -a.



Нетрудно найти экспертов по Git, которые считают необходимым создавать связное повествование о фиксациях. Вместо сообщения вида «добавил индексный объект, а заодно исправил несколько ошибок» такой эксперт создаст две фиксации: одну с сообщением «добавил индексный объект», а другую с сообщением «исправления ошибок». Такая степень контроля существует, потому что по умолчанию в индекс ничего не добавляется, так что автор может добавить ровно столько, чтобы описать в точности одно изменение – запись индекса в объект фиксации, – а затем добавить в чистый индекс новый набор элементов для порождения следующего объекта. Мне как-то встретился блог, в котором автор на нескольких страницах описывал принятую им процедуру фиксации: «В самых сложных случаях я вывожу список различий, просматриваю его и раскрашиваю в шесть цветов...» Однако, пока вы не стали экспертом по Git, такая степень контроля намного превышает ваши потребности. Поэтому отказ от флага -a в команде git commit следует считать продвинутым использованием, большинству людей не нужным. В идеальном мире флаг -a подразумевался бы по умолчанию, но, поскольку это не так, не забывайте про него.

Команда git commit -a добавляет новый объект фиксации в репозиторий, включая в него все изменения, зарегистрированные в индексе, а затем очищает индекс. Сохранив работу, вы теперь можете добавлять новые изменения. Более того – и это величайшее благо, даруемое нам системой управления версиями, – вы можете удалить любой код, будучи уверены, что при необходимости сможете его восстановить. Не засоряйте код большими закомментированными блоками – удаляйте их!



После фиксации вы почти наверняка хлопнете себя по лбу, поняв, что кое-что забыли. Но не торопитесь создавать новую фиксацию, просто выполните команду git commit --amend -a, которая перезапишет последний объект фиксации.

Дуализм дельты и мгновенного списка

Физики иногда рассматривают свет как волну, а иногда как частицу; точно так же объект фиксации в одних случаях лучше представлять себе как полный мгновенный снимок проекта, а иногда как отличие от предшественника (дельту). При любом взгляде он содержит имя автора, имя объекта (как мы увидим ниже), сообщение, заданное во флаге -m, и (если это не самая первая фиксация) указатель на родительский объект (или объекты) фиксации.

Что представляет собой фиксация изнутри: дельту или мгновенный снимок? Может быть, и то, и другое. Было время, когда Git всегда хранил мгновенный снимок, если только с помощью команды git gc (сборка мусора) набор снимков не сжимался в набор дельт. Но пользователи жаловались на необходимость помнить о команде git gc, поэтому теперь она автоматически запускается после определенного числа выполненных команд, так что Git в большинстве случаев (но отнюдь не всегда) хранит дельты.

После создания объекта фиксации ваши действия с ним сводятся в основном к просмотру содержимого. Чтобы увидеть дельты, хранящиеся в объекте фиксации, используется команда git diff, а для вывода метаданных – команда git log.

Из метаданных самым главным является имя объекта, которое представляет собой непрезентабельную, но вполне разумную строку: SHA1-хэш, то есть 40 шестнадцатеричных цифр, гарантирующих, что ни у каких двух объектов не будет

одинакового имени и что один и тот же объект будет иметь одинаковое имя во всех копиях репозитория. В ходе фиксации файлов на экран выводятся несколько первых цифр хэша, а команда git log позволяет получить список всех объектов фиксации в истории, причем каждый объект представлен своим хэшем и сообщением, заданным в момент фиксации (о прочих метаданных можно узнать, выполнив команду git help log). К счастью, достаточно ввести лишь часть хэша, которая уникально идентифицирует фиксацию. Поэтому, посмотрев на историю и решив, что вам нужна ревизия с номером fe9c49cddac5150dc974de1f7248a1c5e3b33e89, вы можете извлечь ее командой:

```
git checkout fe9c4
```

В результате производится раскрутка дельт – то, что несколько хуже, но все же умеет делать patch, – с откатом к состоянию проекта на момент фиксации fe9c4.

В любом объекте фиксации хранятся только указатели на родителей, поэтому если выполнить git log после извлечения старого объекта, то мы увидим лишь объекты на пути к нему, а все более поздние исчезнут. Редко используемая команда git reflog покажет полный перечень объектов фиксации, известных репозиторию, но более простой способ возврата к определенной версии проекта дает *метка* (tag) – понятное человеку имя, которое не нужно искать в истории. Метки – это самостоятельные объекты, хранящиеся в репозитории; в каждой метке запоминается указатель на помеченный объект фиксации. Чаще всего используется метка master, которая ссылается на последний объект фиксации в главной ветви (поскольку мы еще не говорили о ветвлении, то главная ветвь, скорее всего, является в вашем проекте единственной). Таким образом, чтобы вернуться назад во времени к последнему запомненному состоянию, нужно выполнить команду:

```
git checkout master
```

Но обратимся снова к git diff; эта команда показывает, какие изменения были произведены с момента последней зафиксированной ревизии. Ее вывод – это как раз то, что было бы записано в следующий объект фиксации командой git commit -a. Как и в случае автономной утилиты diff, команда git diff > diffs выводит файл, который удобнее просматривать в текстовом редакторе с цветовой подсветкой.

Без аргументов git diff показывает различие между индексом и тем, что находится в каталоге проекта; если вы еще ничего не добавили в индекс, то это будут все изменения с момента последней фиксации. Если задано одно имя объекта фиксации, то git diff показывает последовательность изменений между этим объектом и тем, что находится в каталоге проекта. Если заданы два имени, то показывается последовательность изменений от одной фиксации до другой:

```
git diff                Показать различия между рабочим каталогом и индексом.
git diff 234e2a         Показать различия между рабочим каталогом и заданным объектом фиксации.
git diff 234e2a 8b90ac  Показать различия между двумя объектами фиксации.
```



Существует несколько соглашений об именовании, позволяющих обойтись без ввода шестнадцатеричных цифр. Имя HEAD относится к последней извлеченной фиксации. Обычно это вершина ветви, в противном случае в сообщениях об ошибке git будет встречаться выражение «detached HEAD».

Чтобы сослаться на родителя именованной фиксации, добавьте ~1 в конец имени, чтобы сослаться на деда – добавьте ~2 и т. д. Следовательно, допустимы такие команды:

```
git diff HEAD~4          Сравнить рабочий каталог с состоянием, которое было четыре фиксации назад.
git checkout master~1    Извлечь предшественника головы главной ветви.
git checkout master~     То же, но короче.
git diff b0897~ b8097    Посмотреть, что изменилось в фиксации b8097.
```

Сейчас вы умеете делать следующее:

- Сохранять инкрементные ревизии проекта.
- Получать журналы зафиксированных ревизий.
- Выяснять, что было недавно изменено или добавлено.
- Извлекать более ранние версии, если нужно откатить внесенные изменения.

Наличие системы резервного копирования, организованной настолько, что можно удалять код, будучи уверенным в возможности его восстановления, само по себе способствует повышению вашего мастерства.

Тайник

Объекты фиксации – это опорные точки, от которых берут начало операции Git. Например, Git предпочитает накладывать заплаты относительно объекта фиксации, и вы можете перейти к любой фиксации, но, выйдя из рабочего каталога, который не соответствует фиксации, вы уже не сможете вернуться назад. Если в текущем рабочем каталоге имеются незафиксированные изменения, то Git предупреждает, что вы находитесь не в точке фиксации, и обычно отказывается выполнять операцию. Один из способов вернуться к объекту фиксации состоит в том, чтобы где-то сохранить всю работу, проделанную с момента последней фиксации, откатить проект к последней фиксации, выполнить операцию, а затем – закончив переход или латание – накатить сохраненные действия.

Так мы приходим к идее *тайника* (stash), специального объекта фиксации, во многом аналогичного тому, что дает команда `git commit -a`, но с несколькими дополнительными возможностями, в частности возможностью сохранения всех не поставленных на учет изменений в рабочем каталоге. Вот как выглядит типичная процедура:

```
git stash
# Сейчас код находится в состоянии на момент последней фиксации.
git checkout fe9c4

# Походим здесь.

git checkout master # Или в любой другой фиксации, которая вам нужна

# Сейчас код в состоянии на момент последней фиксации, восстановим потайные изменения:
git stash pop
```

Еще одна альтернатива команде `git checkout` извлечения ранее сохраненной версии в рабочий каталог – команда `git reset --hard`, которая возвращает рабочий каталог в состояние, в котором он был на момент последней команды `checkout`. Команда звучит сурово, потому что так оно и есть: вы собираетесь выбросить на свалку всю работу, проделанную с момента последнего извлечения.

Деревья и их ветви

В репозитории существует единственное дерево, которое создается в момент, когда первый автор нового репозитория выполнил команду `git init`. Надо думать, вы знакомы с древовидными структурами, в которых имеется набор вершин, и каждая вершина связана с некоторым количеством потомков и одним родителем (а в экзотических деревьях типа Git'овского родителей может быть несколько).

Все объекты фиксации, кроме самого первого, имеют родителя, и в каждом объекте хранится дельта между им самим и родительским объектом. Конечный узел последовательности, вершина ветви, помечается именем ветви. Для наших целей можно считать, что имеется взаимно однозначное соответствие между вершинами ветвей и последовательностями дельт, породившими данную ветвь. Наличие такого соответствия означает, что мы можем рассматривать ветвь и объект фиксации на ее вершине как синонимы. Таким образом, если вершиной ветви `master` является объект фиксации `234a3d`, то команды `git checkout master` и `git checkout 234a3d` абсолютно эквивалентны (до тех пор, пока не будет записан новый объект фиксации, который получит метку `master`). Это также означает, что список объектов фиксации на ветви всегда можно восстановить, начав с объекта на вершине и следуя вдоль ветви назад к корню дерева.

Обычно принято располагать полностью работоспособный код на главной ветви. Если вы хотите добавить новую функцию или провести какое-то исследование, создайте новую ветвь. Когда эта ветвь будет отлажена, вы сможете объединить новую функциональность с главной ветвью, а как именно, объясняется ниже.

Существуют два способа создать новую ветвь, расщепив тем самым текущее состояние проекта:

```
git branch newleaf      # Создать новую ветвь...
git checkout newleaf    # затем извлечь только что созданную ветвь.

        # Или выполнить сразу оба шага:
git checkout -b newleaf
```

После создания новой ветви мы можем переключаться между вершинами двух ветвей с помощью команд `git checkout master` и `git checkout newleaf`.

В какой ветви мы сейчас находимся? Легко выяснить с помощью команды

```
git branch
```

которая выведет список всех ветвей и поставит * рядом с активной.

Что случилось бы, если бы вы построили машину времени, вернулись в прошлое раньше момента своего рождения и убили своих родителей? Если верить научной фантастике, то при изменении истории настоящее не изменяется, зато возникает альтернативная история. Поэтому если вы извлечете старую версию, произведете в ней изменения и сохраните новый объект фиксации, содержащий эти изменения, то получите новую ветвь, отличающуюся от главной. Команда `git branch` покажет, что после такого разветвления прошлого вы оказываетесь в ветви `(no branch)`. Непомеченные ветви создают разного рода проблемы, поэтому, обнаружив, что вы

что-то делаете в ветви (no branch), незамедлительно выполните команду git branch -m new_branch_name, чтобы поименовать только что выросшую ветвь.

> **Визуальные средства**
>
> Существует несколько графических интерфейсов, которые особенно полезны для отслеживания расхождения и объединения ветвей. На основе Tk написаны программы gitk и git gui, а команда git instaweb запускает веб-сервер, с которым можно взаимодействовать из браузера. Можете поинтересоваться у менеджера пакетов или у интернет-поисковика, какие еще есть варианты.

Объединение

До сих пор мы генерировали новые объекты фиксации одним из двух способов: создание начального объекта или применение списка дельта из индекса. Ветвь – это тоже последовательность дельт, поэтому, имея произвольный объект фиксации и список дельт из некоторой ветви, мы можем создать новый объект фиксации, применив этот список дельт к существующему объекту. Это и называется *объединением*. Чтобы объединить все изменения, приведшие к появлению узла newleaf, с ветвью master, переключимся на master и выполним команду git merge:

```
git checkout master
git merge newleaf
```

Пусть, например, вы создали ветвь, отходящую от master, для разработки какой-то новой функциональности и вот наконец успешно прогнали все тесты; тогда применение всех дельт, лежащих на ветви разработки, к ветви master должно создать новый объект фиксации, в котором уже присутствует новая функциональность.

Предположим, что во время работы над новой функциональностью вы ни разу не извлекали ветвь master и, стало быть, не вносили в нее никаких изменений. Тогда применение последовательности дельт из другой ветви сведется просто к воспроизведению всех изменений, хранящихся в каждом объекте фиксации в этой ветви. В Git это называется *быстрой перемоткой вперед* (fast-forward).

Но если вы вносили какие-то изменения в ветвь master, то проблема перестает быть такой простой и не сводится к быстрому накату всех дельт. Допустим, к примеру, что в точке разветвления в файле *gnomes.c* была такая строка:

```
short int height_inches;
```

В ветви master вы удалили модификатор, сужающий тип:

```
int height_inches;
```

Целью создания ветви newleaf было изменение единиц измерения:

```
short int height_cm;
```

В этой точке Git оказывается в безвыходном положении. Чтобы понять, как объединить эти две строки, нужно знать, что имел в виду человек. Git в этом случае включает в текстовый файл оба варианта:

```
<<<<<<< HEAD
int height_inches;
```

```
=======
short int height_cm;
>>>>>>> 3c3c3c
```

Объединение откладывается до тех пор, пока вы сами не отредактируете файл, точно указав, что хотели сделать. В этом примере пять строк, оставленные Git, надо полагать, сведутся к одной:

```
int height_cm;
```

Ниже описана процедура объединения в случае, когда быстрая перемотка не применима, то есть изменения имеются в обеих ветвях.

1. Выполните команду `git merge other_branch`.
2. Скорее всего, вы увидите сообщение о наличии конфликтов, требующих разрешения.
3. Проверьте список необъединенных файлов с помощью команды `git status`.
4. Выберите файл для ручной правки. Откройте его в текстовом редакторе, и если это конфликт содержимого, то найдите маркеры конфликтов. Если же конфликт связан с именем или местоположением файла, переместите файл туда, где ему положено быть.
5. Выполните команду `git add your_now_fixed_file`.
6. Повторяйте шаги 3—5, пока не будут устранены все конфликты.
7. Выполните команды `git commit`, чтобы завершить операцию объединения.

Пусть во время этой ручной работы вас утешает мысль, что Git очень осторожен в том, что касается объединения, и ни при каких обстоятельствах не сделает автоматически ничего, что могло бы привести к потере вашей работы.

По завершении операции все изменения, внесенные в боковой ветви, представлены в окончательном объекте фиксации на ветви, указанной в качестве цели объединения, поэтому боковую ветвь обычно удаляют:

```
git delete other_branch
```

Метка `other_branch` удалена, но объекты фиксации, приведшие к ее появлению, по-прежнему находятся в репозитории для справки.

Перемещение

Допустим, в понедельник вы создали новую ветвь для тестирования. Со вторника по четверг вы вносили многочисленные изменения в главную и тестовую ветвь. В пятницу при попытке объединить тестовую ветвь с главной выявилось гигантское количество мелких конфликтов, которые предстоит разрешать.

Отмотаем пленку назад к началу недели. В понедельник вы создали тестовую ветвь, а это означает, что у последних объектов фиксации в обеих ветвях имеется общий предок: понедельничный объект фиксации в главной ветви. Во вторник вы создали новый объект фиксации в главной ветви, скажем `abcd123`. В конце дня вы накатываете все изменения, произведенные в главной ветви, на тестовую:

```
git branch testing   # перейти на тестовую ветвь
git rebase abcd123   # или эквивалентно: git rebase main
```

Команда rebase воспроизводит на тестовой ветви все изменения, произведенные в главной ветви с момента создания общего предка. Возможно, придется вручную выполнить объединение, но это работа всего за один день – все-таки задача более обозримая.

Теперь все изменения вплоть до объекта фиксации abcd123 присутствуют в обеих ветвях, то есть ситуация такая, как будто мы разветвились на этом объекте, а не на понедельничном. Отсюда и название операции: тестовая ветвь перемещена и теперь отходит от другой точки на главной ветви.

Перемещение производится также в конце рабочего дня в среду, четверг и пятницу, причем каждая операция сравнительно безболезненна, поскольку в тестовой ветви уже присутствуют изменения, произведенные в главной на протяжении предшествующих дней.

Операция перемещения часто считается продвинутым использованием Git, поскольку в других системах, не настолько хорошо работающих с накатом дельт, ее нет. Но на практике перемещение и объединение равноценны: в обоих случаях изменения, произведенные в другой ветви, применяются для создания объекта фиксации, а разница заключается только в том, сшиваете вы концы двух ветвей (как в случае объединения) или хотите, чтобы они продолжали существовать раздельно (как в случае перемещения). Типичное использование – переместить дельты с главной ветви на боковую и объединить изменения в боковой ветви с главной, то есть операции в некотором смысле симметричны. Как было указано выше, накопление изменений в обеих ветвях может затруднить окончательное объединение, поэтому рекомендуется выполнять перемещение достаточно часто.

Дистанционные репозитории

Все, о чем шла речь до сих пор, происходило в пределах одного дерева. Если вы клонировали репозиторий, находящийся где-то в другом месте, то сразу после клонирования ваш клон и оригинал имеют одинаковые деревья с идентичными объектами фиксации. Однако если вы и коллеги продолжаете работать, то добавляются различные объекты фиксации.

У вашего репозитория имеется список *дистанционных*[1] *репозиториев*, каждый из которых представляет собой указатель на репозиторий, находящийся где-то на другой машине. Если вы получили свой репозиторий командой git clone, то репозиторий, который вы клонировали, будет в вашем новом репозитории называться origin. В типичном случае это единственный дистанционный репозиторий, с которым вы будете иметь дело.

Впервые после клонирования выполнив команду git branch, вы увидите одну-единственную ветвь вне зависимости от того, сколько ветвей было в оригинале. Но если выполнить команду git branch -a, которая показывает все известные Git

[1] Выбран этот термин, а не «удаленный репозиторий», чтобы не возникало мысли о том, что репозитория больше нет. – *Прим. перев.*

ветви, то вы увидите не только локальные, но и дистанционные ветви. Если вы клонировали репозиторий из Github и т. п., то эта команда позволит узнать, помещали ли другие авторы новые ветви в центральный репозиторий.

Копии ветвей в локальном репозитории пребывают в том состоянии, в каком были на момент первоначального получения. На следующей неделе для обновления дистанционных ветвей информацией из оригинального репозитория выполните команду `git fetch`.

Имея в своем репозитории актуальные дистанционные ветви, вы можете произвести объединение, указав полное имя ветви, например `git merge remotes/origin/master`.

Вместо двухшаговой процедуры `git fetch; git merge remotes/origin/master` можно обновить ветвь одной командой

```
git pull origin master
```

которая скачивает дистанционные изменения и тут же объединяет их с текущим репозиторием.

Обратная операция называется `push`, она позволяет записать в дистанционный репозиторий свою последнюю фиксацию (но не состояние индекса или рабочий каталог). Если вы работаете в ветви с именем `bbranch` и хотите записать ее в дистанционный репозиторий под тем же именем, выполните команду:

```
git push origin bbranch
```

Скорее всего, при записывании изменений применение ваших дельт к удаленной ветви не будет прямой перемоткой (это означало бы, что коллеги ничего не делают). Для разрешения конфликтов объединения требуется участие человека, но на стороне дистанционного репозитория человека, наверное, нет. Следовательно, Git при записи в удаленный репозиторий разрешает только прямую перемотку. А как гарантировать, что ваша операция записи – прямая перемотка?

1. Выполните команду `git pull origin`, чтобы получить изменения, произведенные с момента последнего извлечения.
2. Выполните объединение, как было показано раньше, – в этом случае человеком, разрешающим конфликты, будете вы.
3. Выполните команду `git commit -a -m "dealt with merges"`.
4. Выполните команду `git push origin master`; это получится, поскольку теперь Git должен применить единственную дельту, что можно сделать автоматически.

До сих пор я предполагал, что вы находитесь в локальной ветви `master` и пытаетесь взаимодействовать с дистанционной ветвью `master`. Если имена другие, то указывайте пары имен ветвей, разделенных двоеточием в формате `source:destination`.

```
git pull origin new_changes:master     Объединить дистанционную new_changes с локальной master
git push origin my_fixes:version_2     Объединить локальную ветвь с дистанционной, имеющей другое имя
git push origin :prune_me              Удалить дистанционную ветвь
git pull origin new_changes:           Извлечь в несуществующую ветвь; будет создан объект фиксации
                                       с именем FETCH_HEAD
```

Ни одна из этих операций не изменяет текущую ветвь, но некоторые создают новую, на которую можно перейти, как обычно, командой `git checkout`.

Центральный репозиторий

Несмотря на все рассуждения о децентрализации, самой простой конфигурацией для совместной работы по-прежнему остается центральный репозиторий, который все могут клонировать. Это означает, что у всех авторов один и тот же оригинальный репозиторий. Именно так обычно работает скачивание из Github и Savannah. Для настройки репозитория на такой режим работы выполните команду `git init --bare`, которая говорит, что никто не вправе ничего делать в этом репозитории, а необходимо его предварительно клонировать (такой репозиторий, не имеющий рабочего каталога, называется *голым*). Существуют также некоторые полезные флаги разрешений, например флаг `--shared=group` позволяет всем членам группы (в смысле POSIX) читать и записывать в репозиторий.

Если дистанционный репозиторий не является голым, то вы не можете ничего записать в ветвь, которую извлек владелец репозитория, в противном случае воцарился бы хаос. Если такая необходимость возникнет, попросите коллегу сменить ветвь командой `git branch` и, когда целевая ветвь освободится, произведите в нее запись.

Альтернативно коллега может настроить открытый голый репозиторий и закрытый рабочий. Вы записываете свои изменения в открытый репозиторий, а коллега извлекает их в свой рабочий, когда ему удобно.

Структура репозитория Git не особенно сложна: объекты фиксации, представляющие изменения с момента создания родительского объекта фиксации, организованы в виде дерева, а в индексе собираются все изменения, которые будут сделаны при следующей фиксации. Но эти элементы позволяют хранить несколько версий программы, без опаски удалять файлы, создавать экспериментальные ветви и впоследствии, после успешного тестирования, объединять их с основной, а также объединять работу коллег со своей собственной. Команда `git help` и интернет-поисковик расскажут вам о многих других приемах, позволяющих удобно организовать свою работу.



Мирное сосуществование

Количество языков программирования стремится к бесконечности, и у большинства из них имеется интерфейс с С. В этой коротенькой главе я дам общее представление о процессе и подробно продемонстрирую интерфейс с языком Python.

У каждого языка есть свои традиции организации пакетов и распространения, а это означает, что, написав комбинированный код на С и включающем языке, вы сталкиваетесь со следующей проблемой: как заставить систему сборки пакетов откомпилировать и скомпоновать ваше творение. Это открывает передо мной возможность рассказать о более продвинутых приемах работы с Autotools, в частности об условной обработке подкаталога и добавлении точек подключения для установки.

Динамическая загрузка

Перед тем как переходить к другим языкам, полезно будет остановиться на функциях С, благодаря которым межъязыковой интерфейс вообще возможен: `dlopen` и `dlsym`. Первая открывает динамическую библиотеку, а вторая ищет в ней символ, например статический объект или функцию.

Эти функции описаны в стандарте POSIX. В Windows имеются аналогичные функции, но называются они `LoadLibrary` и `GetProcAddress`; для простоты изложения я ограничусь именами POSIX.

Название «разделяемый объектный файл» точно раскрывает смысл идеи: такой файл содержит список объектов, в том числе функций и статически определенных структур, предназначенных для использования в других программах.

Использование такого файла можно уподобить поиску элемента в текстовом файле, содержащем список элементов. В случае текстового файла мы сначала вызываем функцию `fopen`, которая возвращает описатель файла, а затем функцию, которая производит поиск в файле и возвращает указатель на найденный элемент. В случае же разделяемого объектного файла функция открытия называется `dlopen`, а функция поиска символа – `dlsym`. А волшебство заключается в том, что можно сделать с возвращенным указателем. В случае списка мы имеем указатель на обычный текст и можем производить с ним стандартные операции. Имея же

указатель на функцию, мы можем эту функцию вызвать, а получив указатель на структуру, работать с ней как с уже инициализированным объектом.

Всякий раз как ваша программа на C вызывает функцию, находящуюся в библиотеке, именно так эта функция ищется и используется. Программа, в которой реализована система подключаемых модулей, делает это, чтобы загрузить функции, написанные другими авторами уже после того, как основная программа была развернута пользователем. Скриптовый язык, желающий вызвать написанный на C код, вызывает для этой цели те же самые функции `dlopen` и `dlsym`.

Чтобы продемонстрировать работу с `dlopen` и `dlsym`, в примере 5.1 приведен рудиментарный интерпретатор C, который делает следующее:

- просит пользователя ввести код функции на C;
- компилирует эту функцию и создает разделяемый объектный файл;
- загружает разделяемый объектный файл с помощью `dlopen`;
- получает указатель на функцию с помощью `dlsym`;
- исполняет функцию, которую только что ввел пользователь.

Вот пример запуска:

```
Я готов выполнить функцию. Но сначала вы должны ее написать.
Введите тело функции. Оно должно оканчиваться строкой, содержащей
единственный символ '}'.

>>double fn(double in){
>> return sqrt(in)*pow(in, 2);
>> }
f(1) = 1
f(2) = 5.65685
f(10) = 316.228
```

Пример 5.1 ❖ Программа запрашивает у пользователя код функции, сразу компилирует его и исполняет функцию (dynamic.c)

```
#define _GNU_SOURCE //чтобы stdio.h включил asprintf
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>
#include <readline/readline.h>

void get_a_function(){
   FILE *f = fopen("fn.c", "w");
   fprintf(f, "#include <math.h>\n"                         ❶
              "double fn(double in){\n");
   char *a_line = NULL;
   char *header = ">>double fn(double in){\n>> ";
   do {
      free(a_line);
      a_line = readline(header);                           ❷
      fprintf(f, "%s\n", a_line);
      header = ">> ";
   } while (strcmp(a_line, "}"));
   fclose(f);
```

```
}

void compile_and_run(){
   char *run_me;
   asprintf(&run_me, "c99 -fPIC -shared fn.c -o fn.so");      ❸
   if (system(run_me)!=0){
      printf("Ошибка компиляции.");
      return;
   }

   void *handle = dlopen("fn.so", RTLD_LAZY);                 ❹
   if (!handle) printf("Failed to load fn.so:%s\n", dlerror());

   typedef double (*fn_type)(double);                         ❺
   fn_type f = dlsym(handle, "fn");
   printf("f(1) =%g\n", f(1));
   printf("f(2) =%g\n", f(2));
   printf("f(10) =%g\n", f(10));
}

int main(){
   printf("Я готов выполнить функцию. Но сначала вы должны ее написать.\n"
      "Введите тело функции. Оно должно оканчиваться строкой, содержащей\n"
      "единственный символ '}\n\n");
   get_a_function();
   compile_and_run();
}
```

❶ Эта функция сохраняет введенное пользователем определение функции, добавляя к нему заголовок математической библиотеки (чтобы были доступны функции pow, sin и т. п.) и синтаксически правильное объявление.

❷ Это интерфейс к библиотеке Readline. Мы передаем функции заголовок, она обеспечивает пользователя удобными средствами ввода дополнительной информации и возвращает строку, содержащую все, что пользователь ввел.

❸ Сейчас введенная пользователем функция находится в *c*-файле, и мы можем откомпилировать его. Возможно, вы захотите модифицировать эту строку, задав свои любимые флаги компилятора.

❹ Открываем разделяемый объектный файл для чтения объектов. Флаг позднего связывания говорит, что имена функций следует разрешать только по мере необходимости.

❺ Функция dlsym возвращает void *, поэтому мы должны сообщить информацию о реальном типе.

Это самый системно-зависимый пример во всей книге. Я пользуюсь библиотекой GNU Readline, которая в некоторых системах устанавливается по умолчанию, так как она позволяет безболезненно решить задачу о вводе одной строки текста пользователем. Для вызова компилятора я пользуюсь функцией system, но сами флаги заведомо нестандартны, поэтому их, возможно, придется изменить, чтобы программа заработала в вашей системе.

Ограничения динамической загрузки

Но, не правда ли, было бы лучше подчистить эту программу, добавить директивы #ifdef, чтобы при работе в Windows использовалась функция LoadLibrary (хотя Glib это уже сделала за нас – см. раздел gmodules в документации по Glib), и встроить все это в цикл «читать-выполнить-напечатать»?

К сожалению, это невозможно при использовании dlopen и dlsym. Например, если бы я захотел вытащить из объектного кода одну строку исполняемого кода, что должна была бы искать dlsym? Локальные переменные сразу исключаются, так как dlsym видит только статические переменные, объявленные в исходном коде глобальными на уровне файла. Так что на примере этого полуфабриката уже проявляются ограничения dlopen и dlsym.

Даже если наш взгляд на язык C ограничивается только функциями и глобальными переменными, все равно есть широкий спектр возможностей. Функции могут создавать новые объекты, а глобальной переменной может быть структура, содержащая список указателей на функции или их имена в виде строк, которые потом можно передать dlsym.

Разумеется, вызывающая система должна знать, какие символы искать и как их использовать. В примере выше я постулировал, что функция имеет прототип double fn(double). Для реализации механизма подключаемых модулей автор вызывающей системы мог бы точно специфицировать, какие символы должны присутствовать в модуле и как они будут использоваться. В случае скриптового языка, загружающего произвольный код, автор разделяемого объектного файла должен был бы написать скриптовый код, который правильно вызывает объекты.

Процесс

При использовании функций dlopen и dlsym нужно заботиться об удобствах пользователя:

- функции на C следует писать так, чтобы их было легко вызывать из других языков;
- писать функцию-обертку, которая вызывает C-функцию из включающего языка;
- обработка структур данных на C – можно ли их передавать во включающий язык и обратно?
- компоновка с библиотекой на C. После того как все откомпилировано, мы должны гарантировать, что на этапе выполнения система будет знать, где искать библиотеку.

Писать так, чтобы можно было понять

Ограничения функций dlopen и dlsym предъявляют определенные требования к написанию C-функций, вызываемых из других языков.

- Макросы обрабатываются препроцессором, и в окончательной разделяемой библиотеке от них не остается и следа. В главе 10 мы рассмотрим разно-

образные способы использования макросов, позволяющие упростить работу с функциями в пределах самого C, так что для построения дружелюбного интерфейса даже не понадобится скриптовый язык. Но если нужно скомпоновать библиотеку с кодом, написанным на другом языке, то о макросах придется забыть, и функция-обертка должна будет взять на себя те заботы, которые обычно возлагаются на макрос, вызывающий функцию.

- Включающему языку необходимо сообщить, как следует использовать объекты, полученные от `dlsym`, например переписать объявление функции в виде, понятном включающему языку. Это означает, что для каждого видимого объекта необходима дополнительная работа со стороны включающего языка, а следовательно, желательно сводить количество интерфейсных функций к минимуму. В некоторых C-библиотеках (например, в libXML, см. раздел «libxml и cURL» на стр. 334) имеется набор функций для полного контроля и «простые» функции-обертки, позволяющие в типичных случаях обойтись одним вызовом; если в вашей библиотеке десятки функций, подумайте, не стоит ли предоставить несколько таких простых интерфейсных функций. Лучше иметь пакет для объемлющего языка, предоставляющий только базовую функциональность C-библиотеки, чем пакет, который невозможно сопровождать и который в конечном итоге обязательно поломается.
- В этой ситуации объекты – отличное решение. Если свести к нескольким строкам содержание главы 11, где эта тема обсуждается подробно, то речь идет о том, чтобы в одном файле определить структуру данных и несколько функций для работы с ней, например `struct_new`, `struct_copy`, `struct_free`, `struct_print` и т. д. У хорошо спроектированного объекта количество интерфейсных функций невелико, или, по крайней мере, среди них можно выделить подмножество, достаточное для нужд включающего языка. Как показано в следующем разделе, наличие центральной структуры для хранения данных еще и облегчает программирование.

Функция-обертка

Для каждой C-функции, которую, как вы думаете, может вызвать пользователь, необходима функция-обертка на стороне включающего языка. У этой функции несколько задач.

- Удобство клиента. Пользователи включающего языка, незнакомые с C, не хотят думать о вызывающей системе C. Они ожидают получить какие-то сведения о функциях из справки, а справочная система, скорее всего, ориентирована на функции и объекты включающего языка. Если пользователи привыкли, что функции являются элементами объектов, а вы не подготовили их к такому применению в C, то можно сконфигурировать объект, как это принято во включающем языке.
- Преобразования в обе стороны. Возможно, во включающем языке целые числа, строки и числа с плавающей точкой представлены типами `int`, `char*`

и `double`, но в большинстве случаев между типами С и включающего языка необходимо какое-то преобразование. На самом деле преобразование придется выполнять дважды: первый раз из типа включающего языка в тип С, а затем, после вызова С-функции, нужно будет преобразовать результат обратно в тип включающего языка. Пример для Python будет показан ниже.

Пользователи ожидают, что будут работать с функцией включающего языка, поэтому трудно обойтись без написания функции включающего языка для каждой С-функции, и таким образом количество нуждающихся в сопровождении функций неожиданно удваивается. Неизбежна также избыточность, потому что умолчания, определенные для входных параметров на стороне С, обычно приходится еще раз определять во включающем языке, а списки аргументов, передаваемых из включающего языка, как правило, нужно проверять перед использованием в С-функции. И восставать против этого бессмысленно: придется мириться с избыточностью и перепроверять код на стороне включающего языка при каждом изменении интерфейса на стороне С. Так устроена жизнь.

Контрабанда структур данных через границу

Временно забудем обо всех языках, кроме С; рассмотрим два С-файла, *struct.c* и *user.c*: в первом создается структура данных в виде локальной переменной с внутренней компоновкой, а во втором она используется.

Простейший способ сослаться на данные через границу файла – воспользоваться указателем: *struct.c* выделяет память для структуры и передает указатель на нее в *user.c*, все счастливы. Определение структуры может быть открытым, и в таком случае пользователь может проанализировать данные, расположенные по указателю, и при необходимости изменить их. Поскольку процедуры в файле *user.c* модифицируют данные, на которые ведет указатель, то между данными, видимыми в *struct.c* и *user.c*, не возникает рассогласований.

Наоборот, если бы *struct.c* передал копию данных, а функция в *user.c* внесла бы в нее какие-то изменения, то мы получили бы рассогласование между данными, видимыми в двух файлах. Если мы ожидаем, что полученные данные будут использованы и сразу же отброшены или трактуются как предназначенные только для чтения или что *struct.c* больше не будет интересоваться этими данными, то никакой проблемы в передаче владения данными пользователю нет.

Таким образом, данные, с которыми *struct.c* еще будет работать, нужно передавать по указателю, а для одноразовых данных можно передавать копию.

Но что, если внутреннее строение структуры данных закрыто? На первый взгляд, функция в файле *user.c* получит указатель и не будет знать, что с ним делать. Однако одну вещь она все-таки сделать может: передать указатель обратно функции в файле *struct.c*. И это совершенно обычная практика. Допустим, имеется объект связанного списка, память для которого выделена функцией создания списка (хотя в GLib такой нет), тогда мы можем вызвать `g_list_append` для добавления элементов в список, `g_list_foreach` – для применения некоей операции ко всем элементам списка и т. д. Нужно лишь передавать каждой такой функции указатель на список.

При реализации интерфейса между C и языком, который не знает, как читать структуры C, такая техника называется *непрозрачным*, или *внешним*, *указателем*. Поскольку псевдонимы типов (typedef), объявленные в разделяемом объектном файле, недоступны для `dlsym`, все структуры в C-коде должны быть непрозрачны для вызывающего языка[1]. Как и в случае двух *c*-файлов, нет никакой неясности относительно того, кто владеет данными, и при наличии достаточного количества интерфейсных функций требуемую работу все-таки удастся выполнить. Немалая доля включающих языков располагает явным механизмом для передачи непрозрачных указателей.

Если включающий язык не поддерживает непрозрачных указателей, то все равно возвращайте указатель. Адрес – это целое число, и представление его в таком виде не влечет за собой никакой двусмысленности (пример 5.2).

Пример 5.2 ❖ Мы можем рассматривать адрес указателя как обычное целое число. В самом C для этого мало причин, но для взаимодействия с включающим языком может оказаться необходимо (intptr.c)

```
#include <stdio.h>
#include <stdint.h> //intptr_t

int main(){
   char *astring = "Я нахожусь где-то в памяти.";
   intptr_t location = (intptr_t)astring;        ❶
   printf("%s\n", (char*)location);              ❷
}
```

❶ Гарантируется, что тип `intptr_t` достаточно широк для хранения адреса (C99 §7.18.1.4(1) и C11 §7.20.1.4(1)).

❷ Разумеется, в результате приведения указателя к типу целого теряется вся информация о типе, поэтому мы должны явно восстановить тип указателя. Эта практика чревата ошибками, поэтому применяйте ее только при работе с системами, которые не понимают указателей.

Что может пойти не так? Если диапазон типа целых чисел во включающем языке слишком узкий, то такой подход может привести к ошибке в зависимости от того, где в памяти находятся данные. В подобном случае лучше бы представить указатель в виде строки, а получив строку назад, разобрать ее с помощью функции `strtoll` (преобразование строки в тип `long long int`). Всегда можно найти выход.

Кроме того, мы предполагаем, что указатель не переустанавливается и не освобождается с того момента, когда он впервые передан включающему языку, и до момента следующего запроса из включающего языка. Например, если в C-коде

[1] Встречаются языки, например Julia или Cython, авторы которых не ограничились механизмом dlopen и dlsym, а разработали методы описания структур C на стороне включающего языка, раскрыв тем самым непрозрачные указатели. Людей, пошедших эти путем, лично я считаю героями.

был произведен вызов `realloc`, то включающей программе необходимо передать новый непрозрачный указатель.

Компоновка

Как мы видели, динамическая компоновка с разделяемым объектным файлом – проблема, решаемая функциями `dlopen` и `dlsym` и их аналогами в Windows.

Но часто существует несколько уровней компоновки: что, если вашему C-коду требуется какая-то системная библиотека и, стало быть, компоновка на этапе выполнения (как описано в разделе «Компоновка во время выполнения» выше)? В мире C на этот вопрос есть простой ответ – используйте Autotools для поиска библиотеки в стандартном пути и задавайте правильные флаги компилятора. Если система сборки включающего языка поддерживает Autotools, то и проблем при компоновке с системными библиотеками не будет. Если можно полагаться на наличие pkg-config, то это также может решить проблему. Если же Autotools и pkg-config не подходят, то желаю вам всяческих успехов в деле надежного использования системы установки включающего языка для правильной компоновки с вашей библиотекой. Похоже, среди авторов скриптовых языков еще немало таких, кто полагает, что компоновка одной C-библиотеки с другой – редко встречающаяся частная задача, которую всякий раз нужно решать вручную.

Python как включающий язык

Далее в этой главе мы рассмотрим на примере языка Python практическое воплощение описанных выше принципов для решения уравнения идеального газа (см. пример 10.12). Сама функция нас сейчас не очень интересует, а основное внимание мы уделим созданию пакета для нее. В сетевой документации по Python хватает детальной информации, но примера 5.3 достаточно для иллюстрации работы абстрактных шагов: регистрация функции, преобразование выходных данных из формата включающего языка в формат C, преобразование результатов из формата C в формат включающего языка. Затем мы перейдем к компоновке.

В библиотеке идеального газа есть только одна функция – вычисление давления по температуре, поэтому окончательный пакет будет ничуть не интереснее программы, печатающей строку «Hello, World» на экране. Тем не менее мы сможем запустить интерпретатор Python и выполнить такую программу:

```
from pvnrt import *
pressure_from_temp(100)
```

В первой строке мы загружаем все элементы из пакета `pvnrt` в текущее пространство имен Python. Во второй строке мы вызываем из Python функцию `pressure_from_temp`, которая загружает C-функцию (`ideal_pressure`), выполняющую всю содержательную работу.

Свой рассказ мы начнем с примера 5.3, где показан C-код, в котором Python API применяется для обертывания C-функции и регистрации ее как части пакета Python, который будет собран далее.

Пример 5.3 ❖ Обертка для функции уравнения идеального газа (py/ideal.py.c)

```
#include <Python.h>
#include "../ideal.h"

static PyObject *ideal_py(PyObject *self, PyObject *args){
   double intemp;
   if (!PyArg_ParseTuple(args, "d", &intemp)) return NULL;    ❶
   double out = ideal_pressure(.temp=intemp);
   return Py_BuildValue("d", out);                            ❷
}

static PyMethodDef method_list[] = {                          ❸
   {"pressure_from_temp", ideal_py, METH_VARARGS,
    "Get the pressure from the temperature of one mole of gunk"},
   {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC initpvnrt(void) {
   Py_InitModule("pvnrt", method_list);
}
```

❶ Python передает один объект, в котором перечислены все аргументы функции, – аналог `argv`. В этой строке аргументы читаются в список С-переменных в соответствии с заданными форматными спецификаторами (по аналогии с `scanf`). Если бы в списке было по одному аргументу типа double, строки и integer, то нужно было бы написать: `PyArg_ParseTuple(args, "dsi", &indbl, &instr, &inint)`.

❷ Для возврата результата список типов и возвращаемых С-значений упаковывается в один объект Python.

❸ Оставшаяся часть файла связана с регистрацией. Мы должны построить список методов, реализованных функциями (имя в Python, имя С-функции, соглашение о вызове, одна строка документации). Список должен завершаться элементом `{NULL, NULL, 0, NULL}`. Затем надо написать функцию с именем `initpkgname`, которая будет читать этот список.

На этом примере видно, как в Python преобразуются типы входных параметров и возвращаемых значений (в Python это делается на стороне С, а в ряде других случаев – на стороне включающего языка). Секция регистрации в конце файла также не слишком сложна. А теперь перейдем к задаче компиляции, решение которой может потребовать изрядной изобретательности.

Компиляция и компоновка

В разделе «Создание пакета с помощью Autotools» выше мы видели, что для генерации библиотеки с помощью Autotools необходимы две вещи: создать файл *Makefile.am* из двух строчек и немного подправить файл *configure.ac*, сгенерированный Autoscan. Помимо этого, в Python имеется собственная система сборки Distutils, поэтому мы должны настроить еще и ее, а затем модифицировать файлы Autotools, так чтобы Distutils запускалась автоматически.

Условный подкаталог для Automake

Я решил поместить все относящиеся к Python файлы в подкаталог главного каталога проекта. Если Autoconf найдет средства разработки на Python, то я попрошу его зайти в этот каталог и сделать все необходимое, в противном случае этот каталог нам не понадобится.

В примере 5.4 показан файл *configure.ac*, который проверяет наличие Python и файлов-заголовков для него и в том случае, когда все компоненты найдены, компилирует файлы в подкаталоге *py*. Первые несколько строк, как и раньше, взяты и файла, сгенерированного `autoscan`, с обычными, описанными выше добавлениями. В следующих строках проверяется наличие Python, я их скопировал из документации по Automake. Здесь создается переменная `PYTHON`, содержащая путь к Python, для *configure.ac* создаются переменные `HAVE_PYTHON_TRUE` и `HAVE_PYTHON_FALSE`, а для makefile – переменная `HAVE_PYTHON`.

Если Python или заголовки для него отсутствуют, то переменная `PYTHON` получит значение :, которые мы впоследствии сможем проверить. Если все необходимое в наличии, то в блоке if-then-fi мы просим Autoconf сконфигурировать подкаталог *py* и текущий каталог.

Пример 5.4 ❖ Файл configure.ac file для сборки Python-пакета (py/configure.ac)

```
AC_PREREQ([2.68])
AC_INIT([pvnrt], [1], [/dev/null])
AC_CONFIG_SRCDIR([ideal.c])
AC_CONFIG_HEADERS([config.h])

AM_INIT_AUTOMAKE
AC_PROG_CC_C99
LT_INIT

AM_PATH_PYTHON(,, [:])                          ❶
AM_CONDITIONAL([HAVE_PYTHON], [test "$PYTHON" != :])

if test "$PYTHON" != : ; then                   ❷
AC_CONFIG_SUBDIRS([py])
fi

AC_CONFIG_FILES([Makefile py/Makefile])         ❸
AC_OUTPUT
```

❶ В этих строках проверяется наличие Python. Если он не найден, то в переменную `PYTHON` записывается :, после чего соответственно инициализируется переменная `HAVE_PYTHON`.

❷ Если переменная `PYTHON` установлена, то Autoconf зайдет в подкаталог *py*, иначе проигнорирует его.

❸ В каталоге *py* находится файл *Makefile.am*, из которого нужно сделать makefile, об этой задаче необходимо сообщить Autoconf.



В этой главе вы не раз встретитесь с дополнительными средствами Autotools, в частности с показанным выше макросом `AM_PATH_PYTHON` и целями Automake `all-local` и `install-exec-hook`. По своей природе Autotools – это базовая система (которую я описал в главе 3)

с точками подключения на любой мыслимый случай. Запоминать их все нет смысла, и, как правило, они не выводимы из базовых принципов. Таким образом, если при работе с Autotools встречается необычная ситуация, то лучше всего поискать в руководствах или в Интернете подходящий рецепт.

Мы также должны уведомить Automake о наличии подкаталога, для этого предназначен еще один блок if-then в примере 5.5.

Пример 5.5 ❖ Файл Makefile.am в корневом каталоге проекта с подкаталогом для Python (py/Makefile.am)

```
pyexec_LTLIBRARIES=libpvnrt.la
libpvnrt_la_SOURCES=ideal.c

SUBDIRS=.

if HAVE_PYTHON          ❶
SUBDIRS += py
endif
```

❶ Autoconf создал переменную `HAVE_PYTHON`, и здесь мы ей воспользуемся. Если она существует, то Automake добавит *py* в список каталогов, которые нужно посетить, в противном случае он ограничится только текущим каталогом.

В первых двух строчках говорится, что Libtool должна создать разделяемую библиотеку `libpvnrt`, которая будет установлена туда, где находятся прочие исполняемые файлы Python; в эту библиотеку следует включить результат компиляции исходного файла `ideal.c`. Далее указывается первый дополнительный подкаталог . (то есть текущий каталог). Статическую библиотеку следует строить раньше обертки, позволяющей обращаться к этой библиотеке из Python, и мы добиваемся этого, поставив `.` в начало списка `SUBDIRS`. Далее если переменная `HAVE_PYTHON` определена, то мы с помощью оператора Automake += добавляем в этот список подкаталог *py*.

Мы настроили систему так, чтобы каталог *py* обрабатывался тогда и только тогда, когда средства разработки Python установлены. Теперь перейдем к самому каталогу *py* и посмотрим, как общаются между собой Distutils и Autotools.

Distutils при поддержке Autotools

Дочитав до этого места, вы, надо думать, уже освоились с процедурой компиляции даже сложных программ и библиотек:

- перечислить участвующие файлы (например, с помощью переменной `your_program_SOURCES` в *Makefile.am* или непосредственно в списке `objects`, как в приведенном ранее примере файла makefile);
- задать флаги компилятора (стандартным способом в переменной `CFLAGS`);
- задать флаги компоновщика и дополнительные библиотеки (в переменной `LDLIBS` для GNU Make или переменной `LDADD` для GNU Autotools).

Это три основных шага, и хотя у них много вариаций, принцип достаточно ясен. Я уже продемонстрировал, как соединить эти три части с помощью простого фай-

ла makefile, с помощью Autotools и даже с помощью псевдонимов оболочки. Теперь посмотрим, как они взаимодействуют с Distutils. В примере 5.6 показан файл *setup.py*, управляющий созданием Python-пакета.

Пример 5.6 ❖ Файл setup.py, управляющий созданием Python-пакета (py/setup.py)

```
from distutils.core import setup, Extension

py_modules= ['pvnrt']

Emodule = Extension('pvnrt',          ❶
       libraries=['pvnrt'],           ❷
       library_dirs=['..'],           ❸
       sources = ['ideal.py.c'])

setup (name = 'pvnrt',                ❹
       version = '1.0',
       description = 'pressure * volume = n * R * Temperature',
       ext_modules = [Emodule])
```

❶ Исходные файлы и флаги компоновщика. В строке `libraries` говорится, что компоновщику нужно указать флаг `-lpvnrt`.

❷ В этой строке говорится, что в состав флагов компоновщика нужно включить флаг `-L..`, чтобы компоновщик искал библиотеки в родительском каталоге. Это приходится задавать вручную.

❸ Здесь перечисляются исходные файлы, как в Automake.

❹ Здесь задаются метаданные пакета, которые нужны Python и Distutils.

Спецификация процесса создания пакета в системе Distutils хранится в файле *setup.py*, в котором среди прочего приводятся стандартные сведения о пакете: его имя, номер версии, однострочное описание и т. д. Именно в этом файле мы описываем все три вышеупомянутых элемента.

- Исходные файлы на C, содержащие обертку для включающего языка (в отличие от файлов библиотеки, которыми занимается Autotools), перечисляются в переменной `sources`.
- Python понимает переменную окружения `CFLAGS`. Переменные, определенные в makefile, не экспортируются в программы, вызываемые make, поэтому в *Makefile.am* в каталоге *py* (см. пример 5.7) непосредственно перед вызовом `python setup.py build` устанавливается переменная оболочки `CFLAGS`, в которую копируется значение переменной Autoconf `@CFLAGS@`.
- Система Distutils требует отделять библиотеки от путей к библиотекам. Поскольку список библиотек меняется не слишком часто, то его вполне можно написать вручную, как в нашем примере (не забудьте включить статическую библиотеку, сгенерированную Autotools при обработке корневого каталога). Но вот каталоги на разных машинах различаются, потому-то мы и попросили Autotools сгенерировать для нас переменную `LDADD`. Это жизнь, ничего не поделаешь.

Я решил построить установочный пакет, так чтобы пользователь вызывал Autotools, а затем Autotools вызывал Distutils. Поэтому наш следующий шаг – как-то уведомить Autotools о том, что ему нужно вызвать Distutils.

На самом деле это единственная обязанность Automake в каталоге *py*, поэтому *Makefile.am* в этом каталоге только этой проблемой и занимается. В примере 5.7 показано, что нам нужен один шаг для компиляции пакета и еще один для его установки, причем для каждого шага имеется отдельная цель в makefile. Для компиляции эта цель называется `all-local` и вызывается, когда пользователь набирает `make`; для установки цель называется `install-exec-hook` и вызывается, когда пользователь набирает `make install`.

Пример 5.7 ❖ Настройка Automake для работы совместно с Distutils (py/Makefile.py.am)

```
all-local: pvnrt

pvnrt:
    CFLAGS='@CFLAGS@' python setup.py build

install-exec-hook:
    python setup.py install
```

Теперь у Automake есть все необходимое для генерации библиотеки в корневом каталоге, у Distutils есть вся нужная ей информация в подкаталоге *py*, и Automake знает, в какой момент вызывать Distutils. Стало быть, пользователь может ввести стандартную последовательность команд `./configure;make;sudo make install`, и в результате будет создана как C-библиотека, так и Python-обертка для нее.

Часть II

Язык

В этой части мы подвергнем критическому анализу все, что знаем о языке C.

У этой процедуры две стороны: понять, какие части языка не следует использовать, и разобраться в том новом, что появилось недавно. Некоторые новшества носят чисто синтаксический характер, например возможность инициализировать элементы структуры по имени, другие – это функции, которые написаны за нас и уже стали общеупотребительными, например позволяющие работать со строками относительно безопасно.

Я предполагаю, что читатель знаком с основами C. Начинающим, возможно, стоит предварительно познакомиться с приложением A.

Материал распределен по главам следующим образом.

Глава 6 – руководство для тех, кого приводят в замешательство (или легкое недоумение) указатели.

В главе 7 мы расчистим место для нового строительства. Мы бегло рассмотрим те концепции, которые изложены в стандартных учебниках и которые, на мой взгляд, должны быть опровергнуты или сочтены неактуальными.

Глава 8 – шаг в другом направлении. Здесь мы уделим внимание идеям, которые в типичном учебнике затрагиваются лишь мимоходом или не рассматриваются вовсе.

Глава 9 посвящена строкам, мы обсудим, как работать с ними без головной боли, связанной с выделением памяти и подсчетом символов. Функции `malloc` будет очень одиноко, ведь вы перестанете ее вызывать.

В главе 10 описываются новые синтаксические конструкции, которые позволяют записывать вызовы функций в соответствии со стандартом ISO, передавая в качестве аргументов списки произвольной длины (например, `sum(1, 2.2, [...] 39, 40)`) или именованные необязательные значения (например, `new_person(.name="Joe", .age=32, .sex='M')`). Как и рок-н-ролл, эти синтаксические новации не раз вызволяли меня из трудных положений. Не знай я о них, давно уже отказался бы от C.

В главе 11 мы препарируем идеи объектно-ориентированного программирования. Это гидра о многих головах, и попытка переноса ее на C целиком стала бы подвигом Геракла сомнительной ценности, однако некоторые аспекты парадигмы легко реализуются, когда в этом возникает необходимость.

Возможно, это прозвучит слишком хорошо, чтобы быть правдой, но иногда всего одна строка кода может удвоить, учетверить, а то и еще больше увеличить скорость работы программы. Секрет заключается в параллельных потоках, и в гла-

ве 12 мы рассмотрим три системы, позволяющие превратить однопоточную программу в многопоточную.

Рассмотрев вопрос о принципах структурирования библиотек, мы в главе 13 воспользуемся некоторыми из них для выполнения математических расчетов, для взаимодействия с интернет-сервером по используемому им протоколу, для доступа к базе данных и других не менее сложных задач.



Ваш приятель – указатель

Таким, как он,
Нравятся все эти милые песенки,
Ему нравится подпевать
И стрелять из ружья.
Правда, он не понимает, что все это значит.

— Nirvana «In Bloom»

Подобно песне о музыке или фильму о Голливуде, указатель – это данные, описывающие другие данные. Есть от чего растеряться: вдруг, ни с того ни с сего оказываешься в странном мире, где обитают ссылки на ссылки, псевдонимы, механизмы управления памятью и функция `malloc`. Но судьба-злодейка не совсем безжалостна, она позволяет выделить из хаоса отдельные компоненты. Например, указатели можно использовать как псевдонимы, не думая о `malloc`, которая вовсе не обязана быть такой вездесущей, как ее часто представляют в учебниках 90-х годов. С одной стороны, синтаксис С с его бесконечными звездочками может запутать, но, с другой, он предлагает средства для работы в особо сложных случаях, например указатели на функции.

В этой главе рассматриваются типичные ошибки и недоразумения. Если вы уже давно пишете на С, то использование указателей стало второй натурой, так что можете пропустить эту главу вообще или просмотреть ее по диагонали. Она рассчитана на людей (и имя им – легион), кто чувствует себя неуверенно при работе с указателями.

Автоматическая, статическая и динамическая память

С предлагает три основные модели управления памятью – на две больше, чем в большинстве языков, и на две больше, чем вам хотелось бы знать. А как бонус для читателя я попозже подкину еще две модели памяти (поточно-локальную в разделе «Поточно-локальная память» и проецируемую в разделе «Использование mmap для очень больших наборов данных»).

Автоматическая

Вы объявляете переменную при первом использовании, и она автоматически удаляется при выходе из области видимости. Без ключевого слова `static` любая переменная, объявленная внутри функции, является автоматической. В типичном языке программирования все данные автоматические.

Статическая

Статические переменные находятся в одном и том же месте в течение всего времени работы программы. Размер массива фиксирован в момент создания, но значения в нем могут изменяться (поэтому массив нельзя назвать истинно статическим). Данные инициализируются до начала выполнения функции `main`, поэтому инициализировать их можно только константами, не требующими вычислений. Переменные, объявленные вне функций (в области видимости файла) и внутри функций с ключевым словом `static`, являются статическими. Если вы забудете инициализировать статическую переменную, она будет инициализирована нулями (или значением `NULL`).

Динамическая

Динамическая[1] память неразрывно связана с функциями `malloc` и `free`, именно здесь происходит большинство ошибок нарушения защиты памяти (segfault)[2]. Как раз из-за этой модели памяти Иисус прослезился[3], когда программировал на C. Кроме того, это единственный тип памяти, позволяющий изменять размер массива после его объявления.

Ниже приведена сводка различий между тремя типами памяти. В следующих главах большая их часть станет предметом подробного обсуждения.

	Статическая	Автоматическая	Динамическая
Обнуляется на этапе запуска	+		
Ограничена областью видимости	+	+	
Можно задавать значение во время инициализации	+	+	
Можно инициализировать неконстантным значением		+	
`sizeof` позволяет получить размер массива	+	+	

---

[1] В оригинале употребляется слово «manual» (ручная), но такая терминология непривычна для русскоязычного читателя. – *Прим. перев.*

[2] В стандартах C99 и C11 §6.2.4 память, полученная от функции malloc, называется выделенной (allocated), но я предпочитаю этот термин, потому что так нагляднее различие между динамической памятью и памятью, выделенной в стеке.

[3] Евангелие от Иоанна, стих 11:35. – *Прим. перев.*

	Статическая	Автоматическая	Динамическая
Сохраняется между вызовами функции	+		+
Может быть глобальной	+		+
Размер массива можно задать во время выполнения		+	+
Размер массива можно изменять			+
Иисус прослезился			+

Некоторые из описанных свойств – это то, что нам нужно от переменной, например возможность изменения размера или удобство инициализации. Другие, например возможность задавать значения на этапе инициализации, – технические следствия внутреннего устройства системы памяти. Поэтому, если вам вдруг понадобилась какое-то специальное свойство, например возможность изменения размера массива во время выполнения, поневоле приходится вникать в подробности `malloc` и указателей. Если бы можно было «зачеркнуть всю жизнь и сначала начать», то мы не стали бы неразрывно связывать три набора свойств с тремя наборами технических неудобств. Но жизнь такая, какая есть.

Стек и куча

У любой функции есть область в памяти, *кадр*, где хранится информация о ней, в том числе адрес возврата по завершении и место для хранения всех автоматических переменных.

Когда функция (например, `main`) вызывает другую функцию, то операции в кадре первой функции приостанавливаются, а для вызванной функции создается новый кадр и помещается в *стек* кадров. По окончании работы функции ее кадр выталкивается из стека, а все хранившиеся в этом кадре переменные пропадают.

К несчастью, на размер стека наложены произвольно выбранные ограничения, поэтому стек гораздо меньше памяти общего назначения – в районе 2–3 мегабайтов (так задавались умолчания в Linux на момент написания книги). Этого хватит для хранения всех трагедий Шекспира, так что о размещении массива из 10 000 целых можете не беспокоиться. Но наборы данных бывают куда больше, а из-за ограничений на размер стека приходится подыскивать для них место где-то еще, а это значит – прибегать к помощи `malloc`.

Функция `malloc` выделяет память не в стеке, а в области, которая называется *кучей*. Размер кучи может быть ограничен или не ограничен; при работе на типичном ПК вполне можно считать, что размер кучи примерно равен объему всей доступной памяти.

Детали окружения и реализации обычно в стандарте не упоминаются, а стек кадров – это как раз деталь реализации. Однако в реализации этого аспекта всегда наблюдался широкий консенсус. Таким образом, описание автоматически распределяемых переменных, приведенное в стандарте C, очень напоминает поведение переменных, выделяемых и уничтожаемых в стеке кадров, а описание того, что в этом документе называется «выделенной памятью», – ни дать ни взять поведение памяти, взятой из кучи.

Все это касается размещения данных в памяти. Но есть еще вопрос об объявлении собственно переменных, здесь свои правила.

1. Если переменная типа `struct`, `char`, `int`, `double` или еще какого-то объявлена либо вне функции, либо внутри функции с ключевым словом `static`, то она статическая, в противном случае автоматическая.

2. При объявлении самого указателя также указывается тип памяти, скорее всего, автоматической или статической (см. правило 1). Но при этом указатель может указывать на данные в памяти любого типа: статический указатель – на динамические данные, автоматический указатель – на статические данные, – возможны любые комбинации.

Правило 2 означает, что определить модель памяти только по наличию ключевых слов невозможно. С одной стороны, это хорошо, потому что не приходится иметь дело с одной нотацией для автоматических массивов и с другой – для динамических. С другой стороны, знать, в какой памяти размещены данные, все равно нужно, иначе можно попытаться изменить размер автоматического массива или забыть про освобождение памяти, занятой динамическим массивом.

Различие между указателем на динамическую память и указателем на автоматическую память позволяет ответить на вопрос, который очень часто возникает у начинающих изучать C: в чем разница между `int an_array[]` и `int *a_pointer`?

Встретив такое объявление:

```
int an_array[32];
```

программа выполняет следующие действия:

- отводит в стеке область для размещения 32 целых;
- объявляет переменную `an_array` как указатель;
- связывает этот указатель с только что отведенной областью памяти.

Эта область выделена в автоматической памяти, то есть вы не можете ни изменить ее размер, ни сохранить данные в ней после автоматического уничтожения по выходе из области видимости. Есть и еще одно ограничение – указатель `an_array` нельзя направить куда-то еще. Поскольку переменную `an_array` нельзя «развести» с выделенной для нее областью для хранения 32 целых, то в книге K&R и в стандарте C говорят, что `an_array` *является* массивом.

Несмотря на указанные ограничения, `an_array` – все же указатель на область в памяти, и к нему применимы обычные правила разыменования указателей (обсуждаются ниже).

Встретив же такое объявление:

```
int *a_pointer;
```

программа выполняет только один из вышеописанных шагов:

- объявляет переменную `an_array` как указатель.

Этот указатель не привязан намертво к какому-то адресу в памяти, поэтому его можно перенаправить на любой другой адрес. Допустимы следующие операции:

```
// динамически выделить блок памяти, направив на него указатель a_pointer:
a_pointer = malloc(32*sizeof(int));

// направить указатель на массив, объявленный выше:
a_pointer = an_array;
```

Таким образом, нотационные различия в объявлениях `int an_array[]` и `int *a_pointer` имеют семантический эффект. Но в других ситуациях, например при

объявлении typedef (скажем, для структуры) или функции, различия не столь отчетливы. Например, в следующем объявлении функции:

```
int f(int *a_pointer, int an_array[]);
```

`a_pointer` и `an_array` ведут себя в точности одинаково. Никакая память не выделяется, поэтому разница между указателями на динамическую и автоматическую память чисто теоретическая. С-функция получает копии входных аргументов, а не оригиналы, а копия указателя на автоматическую память не имеет таких ограничений привязки, какие были у исходного массива. Так что когда речь идет об аргументах функции, разницы между массивом и указателем нет вовсе, и в стандартах C99 §6.7.5.3(7) и C11 §6.7.6.3(7) сказано: «объявление параметра вида 'массив типа' заменяется объявлением 'квалифицированный указатель на тип'» (квалификаторы `const`, `restrict`, `volatile` и `_Atomic` сохраняются в ходе преобразования из массив-*типа* в указатель-на-*тип*). В примере выше размер массива не задавался, но ***приведение к указателю*** производится даже в случае объявления вида `int g(int an_array[32])`.

Я выработал у себя привычку всегда использовать форму `*a_pointer` в заголовках функций и в объявлениях typedef, потому что при этом нужно принимать на одно решение меньше и к тому же сохраняется правило о чтении сложных объявлений справа налево (см. раздел «Форма существительное–прилагательное»).



Ваша очередь. Просмотрите какую-нибудь свою программу и определитесь с классификацией: какие данные хранятся в статической памяти, какие в автоматической, а какие в динамической; какие переменные являются автоматическими указателями на динамическую память, на статические значения и т. д. Если под рукой ничего нет, выполните это упражнение для кода из примера 6.6.

Переменные для хранения постоянного состояния

Эта глава посвящена главным образом взаимодействиям между автоматической памятью, динамической памятью и указателями, а статические переменные как-то остались на обочине. Однако будет уместно сделать паузу и рассказать, для чего могут оказаться полезны статические переменные.

У статических переменных может быть локальная область видимости. Это означает, что можно определить переменную, которая видна только в одной функции, но при выходе из этой функции сохраняет свое значение. Это очень удобно для реализации внутренних счетчиков или «нестираемой» области для рабочих данных. Поскольку статическая переменная никогда не перемещается, указатель на нее остается действительным и после выхода из функции.

В примере 6.1 приведен традиционный учебный пример: последовательность чисел Фибоначчи. Ее первый член равен 0, второй – 1, а каждый последующий – сумме двух предыдущих.

Пример 6.1 ❖ **Последовательность чисел Фибоначчи, генерируемая конечным автоматом (fibo.c)**

```
#include <stdio.h>

long long int fibonacci(){
    static long long int first = 0;
    static long long int second = 1;
    long long int out = first+second;
    first=second;
    second=out;
    return out;
}

int main(){
    for (int i=0; i< 50; i++)
        printf("%lli\n", fibonacci());
}
```

Обратите внимание на тривиальность функции `main`. Функция `fibonacci` – это небольшой автомат, который работает сам по себе; `main` остается только дернуть функцию, и та выдаст очередное значение. Иначе говоря, эта функция представляет собой простой *конечный автомат*, а ключом к реализации конечных автоматов в C являются статические переменные.

Как жить таким статическим конечным автоматам в мире, где любая функция должна быть потокобезопасной? Комитет ИСО по языку C предвидел этот вопрос и включил в стандарт C11 тип памяти `_Thread_local`. Стоит включить его в объявление:

```
static _Thread_local int counter;
```

как вы получите свой счетчик в каждом потоке. Подробнее эта тема рассматривается в разделе «Поточная локальность» на стр. 299.

Объявление статических переменных

Статические переменные, даже объявленные внутри функции, инициализируются на этапе запуска программы, до входа в `main`, поэтому их можно инициализировать только константным значением.

```
//это ошибка: нельзя вызывать gsl_vector_alloc() до входа в main()
static gsl_vector *scratch = gsl_vector_alloc(20);
```

Это неприятность, но она легко устраняется с помощью макроса, который вначале присваивает переменной значение 0, а содержательную инициализацию производит при первом использовании:

```
#define Staticdef(type, var, initialization) \
static type var = 0; \
if (!(var)) var = (initialization);

// порядок вызова:
Staticdef(gsl_vector*, scratch, gsl_vector_alloc(20));
```

Это работает, коль скоро `initialization` не равно нулю (или, в случае указателей, значению NULL). В противном случае переменная будет инициализироваться при каждом обращении. Впрочем, может быть, это и хорошо.

Указатели без malloc

Говоря компьютеру *установи A в B*, я могу иметь в виду одно из двух.

- Скопировать значение B в A. Если я затем инкрементирую A с помощью операции A++, то B не изменится.
- Сделать A псевдонимом B. В этом случае операция A++ приводит также к инкременту B.

Всякий раз, когда программа собирается *установить A в B*, необходимо решить, что вы хотите создать: копию или псевдоним. И это никакая не специфика C.

В C мы всегда создаем копию, но если копируется адрес элемента данных, то копия указателя является новым псевдонимом этого элемента. Это изящная реализация псевдонимии.

В других языках свои обычаи: в семействе языков, производных от LISP, преобладает создание псевдонимов, а для копирования имеется специальная команда set; Python обычно копирует скаляры, но создает псевдонимы для списков (если не используется команда copy или deepcopy). Зная, чего ожидать, вы сможете избежать целого класса ошибок.

В библиотеке GNU Scientific Library имеются объекты для представления векторов и матриц, у тех и других есть элемент data, представляющий собой массив чисел типа double. Допустим, что с помощью typedef мы определили пару (вектор, матрица) и объявили массив таких пар:

```
typedef struct {
    gsl_vector* vector;
    gsl_matrix* matrix;
} datapair;

datapair your_data[100];
```

Предположим, что вы часто обращаетесь к первому элементу первой матрицы:

```
your_data[0].matrix->data[0]
```

Если вы хорошо знаете, как увязаны между собой различные части структур, то понять такую запись несложно, но набирать утомительно. Давайте заведем псевдоним:

```
double *elmt1 = your_data[0].matrix->data;
```

В данном случае знак равенства означает создание псевдонима: копируется только указатель, и если мы изменим *elmt1, то данные, упрятанные глубоко в your_data, также изменятся.

Псевдонимы не имеют никакого отношения к malloc и потому являются примером полезного использования указателей вне связи с управлением памятью.

Приведем еще один пример ситуации, когда применение malloc совершенно необязательно. Пусть есть функция, которая принимает указатель:

```
void increment(int *i){
    (*i)++;
}
```

Пользователи этой функции, в сознании которых указатели и `malloc` неразрывно связаны, могут подумать, что для передачи указателя нужно сначала динамически выделить память:

```
int *i = malloc(sizeof(int)); // столько усилий - и все зря
*i = 12;
increment(i);
...
free(i);
```

Однако проще всего положиться на автоматическое выделение:

```
int i=12;
increment(&i);
```



Ваша очередь. Выше я дал совет: всякий раз, говоря ***установи А в В***, думайте, что вы хотите получить: псевдоним или копию. Возьмите какой-нибудь код (все равно, на каком языке) и, анализируя его строка за строкой, ответьте на вопрос, что есть что. Нет ли в этом коде мест, где было бы разумно заменить копию псевдонимом?

Структуры копируются, для массивов создаются псевдонимы

Как видно из примера 6.2, копирование содержимого структуры – операция, записываемая одной строкой.

Пример 6.2 ❖ Нет, копировать структуру поэлементно необязательно (copystructs.c)

```
#include <assert.h>

typedef struct{
   int a, b;
   double c, d;
   int *efg;
} demo_s;

int main(){
   demo_s d1 = {.b=1, .c=2, .d=3, .efg=(int[]){4,5,6}};
   demo_s d2 = d1;

   d1.b=14;           ❶
   d1.c=41;
   d1.efg[0]=7;

   assert(d2.a==0);   ❷
   assert(d2.b==1);
   assert(d2.c==2);
   assert(d2.d==3);
   assert(d2.efg[0]==7);
}
```

❶ Изменим d1 и посмотрим, изменилась ли d2.

❷ Все эти утверждения справедливы.

Как всегда, вы должны понимать, является ли операция присваивания копированием данных или созданием псевдонима. И как же обстоит дело здесь? Мы изменили d1.b и d1.c, но d2 не изменилась, следовательно, это копирование. Но копия указателя по-прежнему указывает на исходные данные, поэтому изменение d1.efg[0] отражается и на d2.efg. Отсюда вывод: если нужно *глубокое копирование*, то есть копирование данных, на которые ведут указатели, то понадобится функция копирования структуры; если же прослеживать указатели не нужно, то функция копирования – излишество, достаточно простого знака равенства.

Для массивов знак равенства копирует лишь псевдоним, но не сами данные. В примере 6.3 мы проделали тот же трюк: сделали копию, изменили оригинал и проверили, что стало с копией.

Пример 6.3 ❖ Структуры копируются, но присваивание одного массива другому приводит к созданию псевдонима (copystructs2.c)

```
#include <assert.h>

int main(){
    int abc[] = {0, 1, 2};
    int *copy = abc;

    copy[0] = 3;
    assert(abc[0]==3);       ❶
}
```

❶ Утверждение справедливо: оригинал изменился вместе с копией.

Пример 6.4 – медленное приближение к катастрофе. Это по существу две функции, которые автоматически выделяют две области памяти: первая выделяет память для структуры, вторая – для небольшого массива. Поскольку память автоматическая, по выходе из той и другой функции выделенная память освобождается.

Функция, которая завершается предложением return x, возвращает значение x вызывающей функции (C99 & C11 §6.8.6.4(3)). Звучит просто, но это значение должно быть скопировано в вызывающую функцию, так как кадр вызванной функции очень скоро будет уничтожен. Как и раньше, в случае структуры, числа и даже указателя вызывающая функция получает копию возвращенного значения, а вот в случае массива – *указатель* на массив, а не копию хранящихся в нем данных.

Тут-то и скрыта ловушка, потому что возвращенный указатель может указывать на область автоматической памяти, выделенной для данных массива, а она при выходе из функции уничтожается. Указатель на освобожденную область памяти хуже, чем бесполезен.

Пример 6.4 ❖ Структуру можно возвращать из функции, массив – нельзя (automem.c)

```
#include <stdio.h>

typedef struct powers {
    double base, square, cube;
```

```
} powers;

powers get_power(double in){                                          ❶
    powers out = {.base = in,
                  .square = in*in,
                  .cube = in*in*in};
    return out;                                                       ❷
}

int *get_even(int count){
    int out[count];
    for (int i=0; i< count; i++)
        out[i] = 2*i;
    return out; // плохо                                              ❸
}

int main(){
    powers threes = get_power(3);
    int *evens = get_even(3);
    printf("threes:%g\t%g\t%g\n", threes.base, threes.square, threes.cube);
    printf("evens:%i\t%i\t%i\n", evens[0], evens[1], evens[2]);       ❹
}
```

❶ Инициализация с помощью позиционных инициализаторов. Если раньше вы с ними не встречались, потерпите еще несколько глав.

❷ Так можно. При выходе создается копия автоматической локальной переменной, после чего оригинал уничтожается.

❸ А так нельзя. Здесь массив трактуется как указатель, поэтому при выходе создается копия указателя. Но после того как автоматически выделенная память будет освобождена, указатель указывает в никуда. Если ваш компилятор достаточно смышленый, то он предупредит об этом.

❹ В функции, из которой была вызвана `get_even`, переменная `evens` – обычный указатель на целое, только указывает он на уже освобожденные данные. Это может привести к нарушению защиты памяти, печати мусора или – если повезло – к печати правильных значений (по чистой случайности).

Если нужна копия массива, то создать ее с помощью одной строки кода все-таки можно, но придется вернуться к явным манипуляциям с памятью, как в примере 6.5.

Пример 6.5 ❖ Для копирования массива нужна функция memmove – допотопная практика, но работает (memmove.c)

```
#include <assert.h>
#include <string.h> //memmove

int main(){
   int abc[] = {0, 1, 2};
   int *copy1, copy2[3];

   copy1 = abc;
```

```
    memmove(copy2, abc, sizeof(int)*3);

    abc[0] = 3;
    assert(copy1[0]==3);
    assert(copy2[0]==0);
}
```

malloc и игрища с памятью

Теперь перейдем к прямой работе с адресами в памяти. Часто эта память выделяется динамически с помощью malloc.

Самый простой способ избежать ошибок, связанных с malloc, – не пользоваться malloc вовсе. Исторически (в 1980-е и 1990-е годы) malloc была необходима для разнообразных манипуляций со строками; в главе 9 будет подробно рассказано о том, как работать со строками, вообще не прибегая к malloc. Еще malloc была нужна для работы с массивами, размер которых устанавливается во время выполнения, – довольно распространенная ситуация; но, как будет видно из раздела «Задание размера массива во время выполнения» ниже, эта практика также устарела.

Вот мой – можно сказать, исчерпывающий – перечень оставшихся причин для использования malloc.

1. Для изменения размера существующего массива необходима функция realloc, но применять ее можно только к областям памяти, первоначально выделенным с помощью malloc.
2. Как объяснялось выше, возвращать массив из функции нельзя.
3. Иногда объекты должны существовать в течение длительного времени после выхода из создавшей их функции. Впрочем, в главе 11 приведено несколько примеров инкапсуляции управления памятью для таких объектов в функциях new/copy/free, так чтобы не замутнять основную логику.
4. Автоматическая память выделяется в стеке, размер которого обычно ограничен несколькими мегабайтами (или того меньше). Поэтому большие участки памяти (порядка мегабайтов) следует выделять в куче, а не в стеке. Но опять же, у вас, наверное, есть функция, которая сохраняет данные в каком-то объекте, и называется она как-то вроде object_new, так что напрямую к malloc вы все равно не обращаетесь.
5. Иногда встречаются функции, которые возвращают указатель. Например, в разделе «Библиотека pthread» на стр. 307 описывается шаблон, согласно которому мы должны написать функцию, возвращающую void *. От этой пули мы уворачиваемся, возвращая просто NULL, но бывает, что деваться некуда. Отметим также, что в разделе «Возврат нескольких значений из функции» описывается, как возвращать структуры, поэтому мы можем вернуть довольно сложное значение без выделения памяти и тем самым обойти еще одну распространенную причину использования malloc внутри функции.

Я составил этот перечень, дабы показать, что не такой уж он и длинный – пункт 5 встречается редко, а пункт 4 зачастую является частным случаем пункта 3, поскольку гигантские наборы данных упаковываются в структуры, напоминающие объекты. В промышленных программах malloc изредка используется, но

обычно обертывается функциями вида new/copy/free, чтобы основному коду не приходилось заниматься деталями управления памятью.

Виноваты звезды

Ну хорошо, мы убедились, что указатели и выделение памяти – вещи разные, но и сама работа с указателями может обернуться проблемой, а все из-за этих чертовых звездочек.

В обоснование имеющегося синтаксиса объявления указателей приводят соображение о том, что объявление и использование выглядят схоже. Имеется в виду, что из объявления

```
int *i;
```

следует, что `*i` – целое, а значит, объявить о том, что `*i` целое, вполне естественно с помощью нотации `int *i`.

Ну и ладно, если вам такое объяснение помогает, то и замечательно. Я не уверен, что смог бы придумать нечто более вразумительное.

Существует простое правило дизайна, пропагандируемое, например, в книге «Дизайн привычных вещей»: *предметы, обладающие совершенно различными функциями, не должны быть похожи* [Norman 2002]. В этой книге в качестве примера приводятся органы управления самолетом – одинаковые рычаги, предназначенные для совершенно разных целей. В критической ситуации это прямой путь к человеческой ошибке.

В этом отношении синтаксис C никуда не годится, потому что `*i` в объявлении и `*i` вне объявления – две большие разницы. Например:

```
int *i = malloc(sizeof(int));   // правильно
*i = 23;                        // правильно
int *i = 23;                    // ошибка
```

Я выбросил из головы правило, согласно которому объявление выглядит как использование. А вместо него сформулировал другое, которое мне помогает: в объявлении звездочка обозначает указатель, вне объявления – значение указателя.

Вот корректно написанный код:

```
int i = 13;
int *j = &i;
int *k = j;
*j = 12;
```

Применяя это правило, легко видеть, что инициализация во второй строке правильна, потому что `*j` находится в объявлении и потому является указателем. В третьей строке `*k` также стоит внутри объявления, поэтому присвоить ей значение `j`, тоже указатель, вполне разумно. В последней строке `*j` находится вне объявления, поэтому обозначает просто целое число, которому можно присвоить значение 12 (и в результате `i` изменится).

Итак, вот первый совет: помните, что `*i` в строке объявления – указатель на что-то, а `*i` в любой другой строке – то, на что указывает указатель.

После разговора об арифметике указателей я дам еще один совет касательно сбивающего с толку синтаксиса объявления указателей.

Все, что нужно знать об арифметике указателей

Элемент массива можно определить с помощью смещения от начала массива. Мы могли бы объявить указатель `double *p`, считать, что это и есть начало массива, а отсчитываемые от него смещения определяют элементы массива: по адресу начала находится первый элемент `p[0]`, на один шаг вперед от начала – второй элемент `p[1]` и т. д. Таким образом, зная указатель и расстояние между двумя соседними элементами, я имею массив.

Можно было бы записать сумму начала и смещения прямо и непосредственно в виде `(p+1)`. Как скажет вам любой учебник, `p[1]` в точности эквивалентно `*(p+1)`, и это объясняет, почему первый элемент массива `p[0] == *(p+0)`. В книге K&R этому вопросу уделено примерно шесть страниц (2-е издание, разделы 5.4 и 5.5).

Из теории вытекает несколько практических правил обозначения массивов и их элементов.

- Объявляйте массив в виде явного указателя `double *p` или в статической либо автоматической форме `double p[100]`.
- В любом случае *n*-ый элемент массива записывается в виде `p[n]`. Не забывайте, что индекс первого элемента равен нулю, а не единице; на него можно сослаться особым образом: `p[0] == *p`.
- Чтобы получить адрес *n*-го элемента (не его значение), пользуйтесь знаком амперсанда: `&p[n]`. Разумеется, для первого элемента имеем `&p[0] == p`.

В примере 6.6 продемонстрированы эти правила в действии.

Пример 6.6 ❖ Простая арифметика указателей (arithmetic.c)

```
#include <stdio.h>

int main(){
    int evens[5] = {0, 2, 4, 6, 8};
    printf("Первое четное число, конечно, равно%i\n", *evens); ❶
    int *positive_evens = &evens[1];                            ❷
    printf("Первое положительное четное число равно%i\n",
           positive_evens[0]);                                  ❸
}
```

❶ Запись `evens[0]` в виде `*evens`.
❷ Адрес элемента 1 сохранен в указателе.
❸ Обычный способ обозначения первого элемента массива.

Я познакомлю вас с одним трюком, основанным на правиле арифметики указателей, согласно которому `p+1` – это адрес следующего элемента массива (то есть `&p[1]`). Это правило, в частности, означает, что при обходе массива в цикле `for` нет нужды в индексе. В примере 6.7 используется указатель, который первоначально направлен на голову списка, а затем пробегает весь массив благодаря выражению `p++`, пока не встретит маркера `NULL`, обозначающего конец массива.

Пример 6.7 ❖ Мы можем воспользоваться тем фактом, что p++ означает «перейти к указателю на следующий», и тем самым упростить запись циклов for (pointer_arithmetic1.c)

```
#include <stdio.h>

int main(){
    char *list[] = {"first", "second", "third", NULL};
    for (char **p=list; *p != NULL; p++){
        printf("%s\n", p[0]);
    }
}
```



Ваша очередь. Как бы вы реализовали этот пример, если бы не знали про p++?

Рассуждения в терминах «начало плюс смещение» дает не слишком много, с точки зрения изящных синтаксических фокусов, зато многое проясняет в плане того, как работает C. Рассмотрим структуру struct. Пусть есть такое определение и объявление:

```
typedef struct{
   int a, b;
   double c, d;
} abcd_s;

abcd_s list[3];
```

Будем считать, что list – это начало отсчета, а list[0].b немного отстоит от него и указывает на b. Иначе говоря, если адрес list можно представить в виде целого (size_t)&list, то b будет расположено по адресу (size_t)&list + sizeof(int), а list[2].d – по адресу (size_t)&list + 6*sizeof(int) + 5*sizeof(double). При таком подходе структура больше напоминает массив с тем отличием, что элементам сопоставляются имена, а не числовые индексы, и размеры и типы элементов различны.

Это не вполне точно из-за *выравнивания*: система может решить, что данные следует хранить в блоках определенного размера и после некоторых полей должно быть оставлено пустое место, чтобы следующие поля начиналось на границе машинного слова или более крупного элемента. Поэтому структура может быть дополнена незначащими байтами в конце, чтобы список структур был правильно выровнен [C99 и C11 §6.7.2.1(15) и (17)]. В заголовке *stddef.h* определен макрос offsetof, который восстанавливает правильность рассуждений в терминах «начало плюс смещение»: поле list[2].d в действительности расположено по адресу (size_t)&list + 2*sizeof(abcd_s) + offsetof(abcd_s, d).

Кстати, дополнение в начале структуры запрещено, поэтому list[2].a находится по адресу (size_t)&list+ 2*sizeof(abcd_s).

Ниже приведена дурацкая функция, которая рекурсивно подсчитывает количество элементов в списке, пока не встретит нулевой элемент. Предположим (отмечу, что это плохая идея), что мы хотели бы воспользоваться этой функцией для

любого списка, в котором нулевое значение имеет смысл, поэтому она будет принимать указатель на void.

```
int f(void *in){
    if (*(char*)in==0) return 1;
    else return 1 + f(&(in[1])); // Это работать не будет.
}
```

Правило «начало плюс смещение» объясняет, почему эта функция не работает. Чтобы обратиться к элементу a_list[1], компилятор должен точно знать длину a_list[0], иначе как понять, насколько смещаться от начала? Но, не видя реального типа, он не может вычислить этот размер.

Многомерные массивы

Один из способов реализовать многомерный массив – создать массив массивов, например int an_array[2][3][7]. Между этим типом и типом int another_array[2][3][6] есть тонкие различия, и на практике это создает больше проблем, чем решает, особенно при написании функций, которые должны работать и с тем, и с другим типом. В учебниках обычно предпочитают рассматривать массивы заведомо фиксированного размера (мы можем ожидать, что в году всегда двенадцать месяцев) или вообще не передавать массив массивов функции.

Послушайтесь меня – забудьте об этом. Учитывать в программах тонкие особенности таких типов – сплошная головная боль. У каждого свои представления о кодировании, но мне такая форма, кроме как в учебниках, почти никогда не встречалась. Гораздо чаще используют представление «начало–размерность–смещение».

Более практичный способ реализовать многомерный массив $N_1 \times N_2 \times N_3$ элементов типа double выглядит так:

- Определить структуру, содержащую один указатель на данные (назовем его data) и список *размерностей*.
- Определить процедуру выделения памяти, которая инициализирует указатель data=malloc(sizeof(double)*N1*N2*N3) и запоминает размерности S1=N1, S2=N2, S3=N3. Еще нам понадобится процедура, которая освобождает выделенную память.
- Определить процедуры get и set: get(x, y, z) получает значение data[x + S1*y + S1*S2*z], а set присваивает значение элементу в той же позиции. При таких процедурах get и set в первом блоке data из S1 элементов будут находиться элементы вида (x, 0, 0). В следующем блоке от S1+0 до S1+S1 находятся элементы вида (x, 1, 0). Строя так одну строку за другой, мы охватим все элементы вида (x, y, 0), для чего понадобится S1*S2 ячеек. Следующая ячейка будет находиться в позиции (0, 0, 1) и т. д., пока не будут исчерпаны все S1*S2*S3 ячеек.

Мы можем проверить, не выходят ли параметры, переданные процедурам get и set, за пределы допустимого диапазона, поскольку запомнили размерности. Значение S3 никогда не используется для доступа к позициям в сетке данных, но запомнить его тем не менее полезно для проверки выхода за границу.

В библиотеке GNU Scientific Library есть хорошая реализация этой идеи для двумерных массивов. Она несколько отличается, так как включает величину первой размерности и маркер смещения. Не составляет никакого труда получать подмножества двумерного массива, например векторы строки или столбца или подматрицы, – нужно только изменить начало и размерности. Для массивов размерности 3 и выше поиск в Интернете даст несколько реализаций описанной здесь системы «начало плюс размерность плюс смещение».

Typedef как педагогический инструмент

Всякий раз конструируя сложный тип, который зачастую включает такие вещи, как указатель на указатель на указатель, спросите себя, не станет ли задача яснее, если воспользоваться псевдонимом типа typedef.

Например, такое популярное определение:

```
typedef char* string;
```

позволяет визуально упростить объявление массива строк и прояснить его назначение. Если вернуться к примеру с p++ в арифметике указателей, можете ли вы положа руку на сердце сказать, что `char *list[]` – вне всякого сомнения, массив строк, а `*p` – строка? В примере 6.8 цикл `for` из примера 6.7 переписан с заменой `char *` на `string`.

Пример 6.8 ❖ Добавление typedef делает громоздкий код чуть более понятным (pointer_arithmetic2.c)

```
#include <stdio.h>
typedef char* string;

int main(){
   string list[] = {"first", "second", "third", NULL};
   for (string *p=list; *p != NULL; p++){
      printf("%s\n", *p);
   }
}
```

Строка объявления `list` теперь читается совсем просто, и из нее совершенно ясно, что это список строк, а объявление `string *p` говорит, что `p` – указатель на строку, то есть `*p` – строка.

Впрочем, помнить, что `string` – это указатель на `char`, все равно нужно, например для того, чтобы убедиться, что `NULL` – допустимое значение.

Можно было бы пойти еще дальше, например объявить двумерный массив строк, дополнив вышеприведенный typedef таким: `typedef stringlist string*`. Иногда это помогает, иногда только перегружает вашу собственную память.

Псевдонимы типов очень хороши для работы с указателями на функции. Если имеется такой заголовок функции:

```
double a_fn(int, int); //объявление
```

то для описания указателя на функцию подобного типа нужно только добавить звездочку (и скобки для учета приоритета операторов):

```
double (*a_fn_type)(int, int); //тип: указатель на функцию
```

Затем поместим в начало `typedef` и тем самым определим новый псевдоним типа:

```
typedef double (*a_fn_type)(int, int); //typedef для указателя на функцию
```

Теперь этот тип можно использовать, как любой другой, например в объявлении функции, принимающей другую функцию в качестве аргумента:

```
double apply_a_fn(a_fn_type f, int first_in, int second_in){
    return f(first_in, second_in);
}
```

Умение определять типы указателей на функции превращает создание функций, принимающих другие функции, из изнурительного экзамена на правильную расстановку звездочек в тривиальное занятие.

Если разобраться, то работа с указателями может оказаться гораздо проще, чем пытаются представить в учебниках, потому что на самом деле это просто адрес или псевдоним, а вовсе не какой-то специальный способ управления памятью. Сложные конструкции типа указателя на указатель на строку, конечно, приводят в замешательство, поскольку наши предки – охотники и собиратели – не выработали навыков для работы с ними. Но благодаря псевдонимам типов С хотя бы дает средства для их приручения.

Глава 7

Несущественные особенности синтаксиса C, которым в учебниках уделяется чрезмерно много внимания

Верю я, что это славно,
Так давай его спалим.

— Porno for Pyros
«Porno for Pyros»

C – быть может, сравнительно простой язык, но его стандарт занимает примерно 700 страниц, поэтому если вы не собираетесь посвятить его изучению всю свою жизнь, важно знать, какие разделы можно без опаски игнорировать.

Начнем с диграфов и триграфов. Если на клавиатуре нет клавиш { и }, то можно использовать вместо них комбинации символов <% и%> (например, `int main()` <% ...%>). Это имело значение в 1990-х годах, когда изготавливались клавиатуры самых разных типов, но сегодня надо еще постараться, чтобы найти клавиатуру без фигурных скобок. Триграфы вида `??<` или `??>`, описанные в стандартах C99 и C11 §5.2.1.1(1), настолько бесполезны, что авторы gcc и clang даже не стали реализовывать их разбор.

Такие темные закоулки языка, как триграфы, игнорировать легко, потому что про них никто и не упоминает. Но другим частям языка в учебниках прошлых лет уделено немало внимания, так как они призваны удовлетворить требования стандарта C89 или преодолевать ограничения оборудования XX века. Но ограничения снимаются, и код становится проще. Если вы получаете удовольствие от выбрасывания лишнего кода и устранения зависимостей, то эта глава для вас.

Ни к чему явно возвращать значение из main

Для разогрева давайте-ка сократим все написанные нами программы на одну строку.

В любой программе должна быть функция `main`, возвращающая значение типа `int`, поэтому строка вида

```
int main(){ ... }
```

является неотъемлемым атрибутом программы.

Наверное, вы полагаете, что раз так, то обязательно должно быть и предложение `return`, в котором указано то самое целое число, которое возвращает `main`. Однако стандарт C гласит: «...если выполнение программы доходит до закрывающей скобки }, которая завершает функцию main, то возвращается значение 0» (C99 и C11 §5.1.2.2(3)). Иначе говоря, если не написать `return 0;` в последней строке `main`, то это будет подразумеваться по умолчанию.

Напомним, что после завершения программы возвращенное ей значение можно увидеть с помощью команды `echo $?`; воспользуйтесь этим, чтобы убедиться, что по достижении конца `main` действительно возвращается 0.

Выше я уже приводил такой вариант *hello.c*, и теперь вы понимаете, как мне удалось свести `main` к одной директиве `#include` и одной строке кода[1]:

```
#include <stdio.h>
int main(){ printf("Hello, world.\n"); }
```



Ваша очередь. Просмотрите свои программы и удалите предложение `return 0` в конце `main`. Изменилось ли что-нибудь?

Пусть объявления текут свободно

Вспомните, когда вы в последний раз читали пьесу. В начале текста перечисляются «действующие лица». Список имен персонажей, наверное, мало что значил для вас, пока вы не начали читать; я, например, всегда пропускаю его и перехожу прямо к началу пьесы. Если, запутавшись в сюжете, вы забудете, кто такой Бенволио, то можно будет вернуться в начало и прочитать строчку, где о нем говорится (друг Ромео, племянник лорда Монтекки). Но это только потому, что вы читаете бумажное издание. А если текст отображается на экране, то можно было бы поискать первое вхождение строки «Бенволио».

---

[1] Кстати говоря, в этом фрагменте демонстрируется еще один способ нажимать на клавиши меньше, чем того требует старый стандарт. Во втором издании K&R объявления вида int main(), в которых не было ничего, кроме скобок, были названы «старым стилем». Такое объявление означало, что о параметрах нет вообще никакой информации, а не информацию о том, что функция не принимает параметров. По старым правилам последнее утверждение нужно было выражать в виде int main(void), а не int main(). Но, начиная с 1999 года, действует правило: «Пустой список в объявителе функции, являющийся частью определения этой функции, означает, что у функции нет параметров» (C99 §6.7.5.3(14) и C11 §6.7.6.3(14)).

Короче говоря, список действующих лиц не очень полезен читателю. Было бы лучше знакомиться с персонажами по мере их появления.

Мне довольно часто встречается примерно такой код:

```
#include <stdio.h>

int main(){
   char *head;
   int i;
   double ratio, denom;

   denom=7;
   head = "Это цикл для деления чисел на семь.";
   printf("%s\n", head);
   for (i=1; i<= 6; i++){
      ratio = i/denom;
     printf("%g\n", ratio);
   }
}
```

В нем три или четыре строки вводного материала (решайте сами, считать ли пустую строку), за которым следует собственно логика.

Это отголоски стандарта ANSI C89, в котором требовалось, чтобы все объявления находились в начале блока из-за технических ограничений ранних компиляторов. Мы по-прежнему должны объявлять переменные, но можем облегчить возлагаемое на автора и читателя бремя, разрешив делать это при первом использовании.

```
#include <stdio.h>

int main(){
   double denom = 7;
   char *head = "Это цикл для деления чисел на семь.";
   printf("%s\n", head);
   for (int i=1; i<= 6; i++){
      double ratio = i/denom;
      printf("%g\n", ratio);
   }
}
```

Здесь объявления встречаются там, где необходимо, поэтому тяжесть бремени сводится к указанию имени типа при первом использовании переменной. Если ваш редактор поддерживает цветовую подсветку синтаксиса, то объявления по-прежнему легко найти (а если не поддерживает, то вы много теряете – и ведь таких редакторов десятки, есть из чего выбрать!).

Когда я читаю незнакомый код и вижу какую-то переменную, то мое первое побуждение – вернуться и посмотреть, где она объявлена. Если переменная объявляется при первом использовании или в строке, непосредственно предшествующей первому использованию, то я сэкономлю несколько секунд. Кроме того, следуя правилу о том, что область видимости переменных должна быть как можно уже, мы существенно уменьшим количество активных переменных, объявленных в

предыдущих строках, а когда функция длинная, это может иметь значение. И последнее – если переменные объявлены непосредственно в цикле, то цикл легче поддается распараллеливанию в OpenMP (см. главу 12).

В данном случае объявления находятся в начале соответствующего блока, а за ними идут строки, не являющиеся объявлениями. Но это просто особенность конкретного примера, а вообще-то объявления и не-объявления могут следовать в произвольном порядке.

Я оставил объявление переменной `denom` в начале функции, но можно было бы и его перенести внутрь цикла, потому что только в цикле она и используется. Мы можем доверять компилятору – он не будет тратить временя и ресурсы на создание и уничтожение переменной на каждой итерации цикла (хотя теоретически именно это он и делает – см. C99 и C11 §6.8(3)). Что касается индекса, то он нужен лишь для управления циклом, так что естественно сократить его область видимости до этого цикла.

Не замедлит ли новый синтаксис работу программы?

Нет.

На первом шаге компиляции код преобразуется во внутреннее языково-независимое представление. Именно по этой причине компиляторы из набора gcc (GNU Compiler Collection) способны генерировать совместимые объектные файлы для языков C, C++, ADA и FORTRAN – по завершении шага синтаксического анализа все они выглядят одинаково. Таким образом, грамматические удобства, включенные в стандарт C99 с целью сделать код понятнее человеку, к моменту порождения исполняемого файла уже не видны.

Да и устройство, на котором выполняется программа, не видит ничего, кроме машинных команд, ему безразлично, какому стандарту соответствовал исходный код: C89, C99 или C11.

Устанавливайте размер массива на этапе выполнения

В том же ключе, что и помещение объявлений в наиболее подходящее место, следует рассматривать и задание длины массива во время выполнения – на основе вычислений, выполненных до его объявления.

Так было не всегда: четверть века назад нужно было либо задавать размер массива на этапе компиляции, либо использовать `malloc`[1].

---

[1] В стандарте C99 требуется, чтобы совместимый компилятор поддерживал массивы переменной длины (variable-length array – VLA). Стандарт C11 сделал шаг назад и объявил такую поддержку факультативной. Лично мне такое поведение комитета по стандартизации кажется неподобающим, потому что обычно комитет делает все возможное для того, чтобы существующий код (даже триграфы!) можно было без ошибок откомпилировать и в будущем.
Коль скоро массивы переменной длины – факультативная часть стандарта, уместно поинтересоваться, насколько они надежны. Авторы компиляторов в борьбе за рынок стремятся к тому, чтобы их компилятор справлялся с как можно большим количеством существующих программ, поэтому неудивительно, что все крупные игроки на этом рынке, всерьез претендующие на совместимость с C11, поддерживают VLA. Даже если вы пишете код для микроконтроллера Arduino (который не является традиционной системой со стеком и кучей) с помощью компилятора AVR-gcc, он все равно может работать с VLA. Я считаю, что код, в котором используются массивы переменной длины, надежен на достаточно широком множестве платформ, а в будущем их число будет только расти.
Читатели, желающие, чтобы их код был совместим с компилятором, не поддерживающим VLA, а во всем остальном стандартным, могут воспользоваться макросом проверки возможностей (см. раздел «Макросы проверки» ниже).

В качестве реального примера, с которым я однажды столкнулся, рассмотрим создание множества потоков, причем их количество пользователь задает в виде аргумента командной строки. Автор решил эту задачу, получая размер массива в виде выражения atoi(argv[1]) (то есть первый аргумент из командной строки преобразуется в целое число), а затем выделяя память для массива такого размера.

```
pthread_t *threads;
int thread_count;
thread_count = atoi(argv[1]);
threads = malloc(thread_count * sizeof(pthread_t));
...
free(threads);
```

Но то же самое можно сделать проще и короче:

```
int thread_count = atoi(argv[1]);
pthread_t threads[thread_count];
...
```

Получается меньше мест, где можно допустить ошибку, и читается такой код как объявление массива, а не как инициализация машинных регистров. Динамически выделенный массив нужно впоследствии освобождать с помощью функции free, а автоматический можно просто бросить на пол в полной уверенности, что его уберут, когда программа выйдет из текущей области видимости.

Меньше приведений

В 1970–1980-е годы функция malloc возвращала указатель типа char*, который нужно было приводить к требуемому типу (если только не выделялась память для строки) следующим образом:

```
//забудьте об этой избыточности
double* list = (double*) malloc(list_length * sizeof(double));
```

Теперь так делать необязательно, потому что malloc нынче возвращает указатель на void, который компилятор сам приведет к нужному типу. Простейший способ выполнить приведение – объявить переменную подходящего типа. Например, функции, которые вынуждены принимать указатель на void, обычно имеют такой вид:

```
int use_parameters(void *params_in){
    param_struct *params = params_in;   //По существу, приводим указатель на void
    ...                                 //к указателю на param_struct.
}
```

В общем случае допустимо присваивать элемент одного типа элементу другого, а C выполнит приведение типов самостоятельно. Если для какого-то конкретного типа это невозможно, то вам в любом случае придется написать функцию преобразования. В C++ это не так, поскольку система типов в этом языке строже и приведения должны быть явными.

Остается две причины для использования синтаксиса приведения типов переменных в C.

Во-первых, при делении целого числа на целое возвращается целый результат, поэтому оба следующих утверждения истинны:

```
4/2 == 2
3/2 == 1
```

Второй случай – источник многочисленных ошибок. Но поправить дело легко: если i целое, то i + 0.0 – число с плавающей точкой с таким же математическим значением, как у целого. Правда, нужно помнить о скобках, если они необходимы, но, в принципе, проблему можно считать решенной. В случае констант 2 – целое, а 2.0 и даже просто 2. – число с плавающей точкой. Поэтому все приведенные ниже варианты работают:

```
int two=2;
3/(two+0.0) == 1.5
3/(2+0.0) == 1.5
3/2.0 == 1.5
3/2. == 1.5
```

Можно использовать и приведение типов:

```
3/(double)two == 1.5
3/(double)2 == 1.5
```

Мне больше нравится вариант с прибавлением нуля, по чисто эстетическим причинам; вы, если хотите, можете выбрать вариант с приведением к типу double. Но не забывайте делать то либо другое всякий раз, как встречаете знак /, потому что пренебрежение этой мудростью – источник многих и многих ошибок (и не только в C; немало и других языков, которые твердо уверены, что int / int => int – несмотря ни на что).

Во-вторых, индексы массива должны быть целыми числами. Это закон (C99 и C11 §6.5.2.1(1)), и компилятор обычно ругается, если в качестве индекса использовать число с плавающей точкой. Поэтому придется привести его к типу целого, даже если вы точно знаете, что в конкретной ситуации индекс и так будет принимать только целочисленные значения.

```
4/(double)2 == 2.0            // Это число с плавающей точкой, а не целое.
mylist[4/(double)2]           // Отсюда и ошибка: индекс с плавающей точкой

mylist[(int)(4/(double)2)]    // Работает. Будьте внимательны со скобками

int index=4/(double)2         // Так тоже работает и к тому же понятнее
mylist[index]
```

Как видите, существуют вполне оправданные причины для приведения типов, но даже в этих случаях есть возможность избежать приведения: прибавить 0.0 и объявить целочисленную переменную для использования в качестве индекса массива.

И дело не только в том, что в программе становится меньше беспорядка. Компилятор проверяет типы и при необходимости выдает предупреждения или со-

общения об ошибках, а явное приведение означает: *оставь меня в покое, я знаю, что делаю*. Рассмотрим, к примеру, следующую коротенькую программу, которая пытается выполнить действие list[7]=12, но дважды совершает классическую ошибку: использует указатель вместо значения, на которое он указывает.

```
int main(){
   double x = 7;
   double *xp = &x;
   int list[100];

   int val2 = xp;          // Clang предупреждает, что указатель используется как int
   list[val2] = 12;
   list[(int)xp] = 12;  // Clang не выдает никаких предупреждений
}
```

Перечисления и строки

Перечисления – отличная идея, зашедшая не туда.

Достоинства очевидны: целые числа лишены мнемоничности, поэтому, собираясь использовать в программе короткий перечень целых чисел, лучше бы дать им какие-нибудь имена. Вот совсем плохой способ решения проблемы без использования ключевого слова enum:

```
#define NORTH 0
#define SOUTH 1
#define EAST 2
#define WEST 3
```

С помощью enum мы можем свести это к одной строке кода, и отладчик правильно поймет, что такое EAST. Следующая строка определенно лучше последовательности директив #define:

```
enum directions {NORTH, SOUTH, EAST, WEST};
```

Но теперь у нас появилось пять новых символов в пространстве имен: directions, NORTH, SOUTH, EAST и WEST.

Чтобы перечисление было полезным, оно обычно должно быть глобальным (то есть объявлять его нужно в заголовке, который включается во многие файлы проекта). Так, часто для перечислений заводится typedef в открытом заголовке библиотеки. Чтобы снизить вероятность возникновения конфликта имен, авторы библиотек заводят имена вида G_CONVERT_ERROR_NOT_ABSOLUTE_PATH или CblasConjTrans – чуть покороче.

И в этот момент безобидную и разумную идею можно отправлять на помойку. Не хочу я набирать такие безобразия, а использую я их настолько редко, что вынужден каждый раз вспоминать, как они пишутся (особенно это относится к редко встречающимся кодам ошибок или флагам – много дней проходит от одного употребления до другого). К тому же изобилие заглавных букв воспринимается как вопль.

Сам я предпочитаю одиночные символы – транспонирование я обозначаю буквой 't', а ошибку в пути – буквой 'p'. Я считаю, что для мнемоничности этого

достаточно – ну действительно, куда больше шансов запомнить, как пишется 'p', чем это заглавная абракадабра, – к тому же в пространстве имен не появляется ни одного нового символа.

Я считаю, что на этом уровне соображения удобства работы важнее эффективности, но при всем при том не стоит забывать, что элемент перечисления – обычно целое число, тогда как char в C занимает один байт. Поэтому при сравнении элементов перечисления приходится сравнивать числа, содержащие 16 или более бит, а при сравнении char – только 8. Так что если быстродействие – для вас значимый довод, то и тогда он свидетельствует не в пользу перечислений.

Иногда приходится комбинировать флаги. Открывая файл с помощью системного вызова open, мы зачастую передаем RDWR|O_CREAT, то есть поразрядное объединение двух элементов перечисления. Возможно, вы редко вызываете open напрямую, а чаще пользуетесь функцией fopen, более дружественной к пользователю. Вместо перечисления ей передается строка из одной или двух букв, например "r" или "r+", которая говорит, что файл нужно открыть для чтения, для записи, для того и другого и т. д.

В этом контексте вы понимаете, что "r" означает *read* (чтение), и эта мнемоника запомнится через два-три употребления fopen, тогда как в приведенном выше примере я каждый раз должен вспоминать, как правильно: CblasTrans, CBLASTrans или CblasTranspose.

К плюсам перечислений нужно отнести небольшой фиксированный набор символов, что позволяет компилятору диагностировать ошибку в случае опечатки и заставить вас внести исправление. При использовании строк вы не узнаете об опечатке до этапа выполнения. С другой стороны, набор строк не фиксирован, поэтому расширять его проще. Так, мне однажды встретился обработчик ошибок, который предлагали использовать в других системах – при условии, что новая система генерирует только те ошибки, которые прописаны в перечислении исходной системы. Если бы коды ошибок были короткими строками, расширение их множества оказалось бы тривиальным делом.

Для использования перечислений есть причины: иногда имеется массив, который нет смысла превращать в структуру, но его элементы тем не менее именованы, а при программировании на уровне ядра давать имена комбинациям битов просто обязательно. Но в тех случаях, когда перечисления служат для задания короткого списка вариантов или кодов ошибок, одиночный символ или короткая строка оказываются ничуть не хуже, но без загромождения пространства имен и памяти программиста.

Метки, goto, switch и break

В старые добрые дни, когда программы писали на ассемблере, не было таких современных удобств, как циклы while и for. А были только условия, метки и команды перехода. Там, где мы написали бы while (a[i] < 100) i++;, нашим предкам приходилось писать нечто вроде:

```
label 1
if a[i] >= 100
   go to label 2
increment i
go to label 1
label 2
```

Если у вас ушла целая минута, чтобы понять, что здесь происходит, подумайте, как это выглядело бы в реальной ситуации, когда циклы перемежаются другими командами, являются вложенными или прерываются досрочно. Мой личный печальный и болезненный опыт говорит о том, что проследить поток выполнения такой программы практически невозможно, и именно поэтому команда `goto` в наши дни считается вредной [Dijkstra 1968].

Вы сами видите, насколько полезным оказалось бы ключевое слово `while` из языка С любому, кто вынужден целыми днями писать на ассемблере. Однако же в С до сих пор существует подмножество, в основе которого лежат метки и переходы, сюда относятся собственно метки, а также ключевые слова `goto`, `switch`, `case`, `default`, `break` и `continue`. Лично я думаю, что эта часть С – переход от приемов написания кода на ассемблере к более современному стилю. Ниже мы рассмотрим эти конструкции и поговорим о том, когда они могут быть полезны. Вместе с тем надо отметить, что все это подмножество языка факультативно в том смысле, что можно написать эквивалентный код, не прибегая ни к одному из этих ключевых слов.

К вопросу о goto

Строка кода на С может быть помечена, для этого перед ней нужно поставить идентификатор и двоеточие. Затем к этой строке можно перейти с помощью предложения `goto`. В примере 7.1 показана простая функция, иллюстрирующая эту идею, – метка строки называется `outro`. Функция вычисляет сумму всех элементов двух массивов при условии, что они отличны от NaN («не число»; см. раздел «Пометка недопустимых числовых значений с помощью маркера NaN» на стр. 169). Если какой-то элемент равен NaN, то это считается ошибкой, и мы должны выйти из функции. Однако при любом способе выхода нужно предварительно освободить оба вектора. Можно было бы включить код очистки трижды (когда в `vector` встречается NaN, когда в `vector2` встречается NaN и когда все хорошо), но правильнее оставить один фрагмент кода и переходить на него по мере необходимости.

Пример 7.1 ❖ Использование goto для чистой реализации выхода в случае ошибки

```
/* Суммирование до обнаружения первого элемента NaN в векторе.
   Если выход нормальный, код ошибки равен 0, если встретился NaN, то 1.*/
double sum_to_first_nan(double* vector, int vector_size,
                        double* vector2, int vector2_size, int *error){
   double sum=0;
   *error=1;
   for (int i=0; i< vector_size; i++){
      if (isnan(vector[i])) goto outro;
      sum += vector[i];
```

```
    }

    for (int i=0; i< vector2_size; i++){
        if (isnan(vector2[i])) goto outro;
        sum += vector2[i];
    }
    *error=0;

    outro:
    printf("The sum until the first NaN (if any) was%g\n", sum);
    free(vector);
    free(vector2);
    return sum;
}
```

Предложение goto работает только внутри функции. Для перехода из одной функции в другую необходим совершенно другой механизм – смотрите документацию по функции longjmp из стандартной библиотеки.

За одиночным переходом проследить сравнительно несложно, он может даже упростить программу, если используется умеренно и обоснованно. Даже Линус Торвальдс, автор ядра Linux, рекомендует ограниченно применять goto для таких целей, как выход из функции в случае ошибки, как в примере выше, или для досрочного завершения обработки. Кроме того, когда в главе 12 мы будем изучать библиотеку OpenMP, выяснится, что из середины распараллеленного блока возврат не допускается. Поэтому чтобы остановить выполнение, нужно либо писать много предложений if, либо перейти в конец блока с помощью goto.

Поэтому, дополняя общепринятое мнение, скажем, что в общем случае goto лучше избегать, но в то же время это идиоматический способ обработки ошибок, который зачастую оказывается чище альтернативных вариантов.

Для тех, кто хотел бы уйти подальше

Предложение goto полезно для выполнения сравнительно простых операций очистки при выходе из функции в случае ошибки. На глобальном уровне существуют три функции «перехода к выходу»: exit, quick_exit и _Exit, а для регистрации операций очистки можно воспользоваться функциями at_exit и at_quick_exit (C11 §7.22.4).

Где-то в начале программы вызывается функция at_exit(fn), чтобы зарегистрировать fn для вызова из exit перед закрытием потоков и завершением программы. Например, если необходимо закрыть сеанс работы с базой данных или сетевое соединение либо требуется закрыть все еще открытые элементы XML-документа, то можно зарегистрировать функцию, которая это сделает. Она должна иметь прототип void fn(void), то есть любая входная информация должна передаваться ей через глобальные переменные. После того как все зарегистрированные функции будут вызваны (в порядке, обратном регистрации), закрываются открытые потоки и файлы и программа завершается.

Совершенно другой набор функций можно зарегистрировать с помощью at_quick_exit. Эти функции (а не те, что были зарегистрированы с помощью at_exit) вызываются при обращении к quick_exit. При таком варианте выхода буферы не сбрасываются и потоки не закрываются.

Наконец, функция _Exit завершает программу максимально быстро: зарегистрированные функции не вызываются, и буферы не сбрасываются.

В примере 7.2 приведена простая программа, которая печатает различные сообщения в зависимости от того, какую не возвращающую управление функцию вы раскомментируете.

Пример 7.2 ❖ **Оставь надежду всяк входящий в функцию, помеченную спецификатором `_Noreturn` (noreturn.c)**

```
#include <stdio.h>
#include <unistd.h> //sleep
#include <stdlib.h> //exit, _Exit и др.

void wail(){
   fprintf(stderr, "OOOOooooooo.\n");
}

void on_death(){
   for (int i=0; i<4; i++)
      fprintf(stderr, "I'm dead.\n");
}

_Noreturn void the_count(){          ❶
   for (int i=5; i --> 0;){
      printf("%i\n", i); sleep(1);
   }

   //quick_exit(1);                  ❷
   //_Exit(1);
   exit(1);
}

int main(){
   at_quick_exit(wail);
   atexit(wail);
   atexit(on_death);
   the_count();
}
```

❶ Ключевое слово `_Noreturn` сообщает компилятору, что не нужно подготавливать для функции выходную информацию.
❷ Раскомментируйте, чтобы посмотреть, что вызывают другие функции выхода.

Предложение switch

Ниже приведен канонический фрагмент кода работы со стандартной функцией `getopt`, предназначенной для разбора аргументов в командной строке:

```
char c;
while ((c = getopt(...))){
  switch(c){
      case 'v':
          verbose++;
          break;
      case 'w':
          weighting_function();
          break;
      case 'f':
          fun_function();
          break;
  }
}
```

Когда c == 'v', увеличивается уровень детализации, когда c == 'w', вызывается весовая функция и т. д.

Обратите внимание на изобилие предложений break (которые приводят к выходу из предложения switch, а не из цикла while – тот продолжает выполняться). Предложение switch просто осуществляет переход к нужной метке (вспомните, что двоеточие как раз и обозначает метку), после чего выполнение программы продолжается с этого места. Таким образом, если бы после verbose++ не было break, то программа радостно выполнила бы weighting_function и пошла бы дальше. Это называется *проваливанием*. Иногда проваливание желательно, но лично мне это всегда казалось платой за элегантность синтаксиса switch-case, по сравнению с метками и goto. В работе [van der Linden 1994, стр. 37–38] по результатам исследования большого объема кода показано, что проваливание оправдано только в 3% случаев.

Если риск внести тонкую ошибку, позабыв поставить break или default, на ваш взгляд, слишком велик, то есть простое решение: не пользуйтесь switch.

Альтернативой предложению switch является последовательность if и else:

```
char c;
while ((c = getopt(...))){
   if (c == 'v') verbose++;
   else if (c == 'w') weighting_function();
   else if (c == 'f') fun_function();
}
```

Она избыточна, потому что повторяется ссылка на c, но в то же время короче, так как не нужно в каждой третьей строке ставить break. Поскольку в таком виде это уже не тонкая обертка вокруг ничем не прикрытых меток и переходов, то и ошибку сделать труднее.

Нерекомендуемый тип float

Математические операции над числами с плавающей точкой могут вызвать проблемы в самых неожиданных местах. Легко придумать алгоритм, который вносит погрешность величиной всего 0,01% на каждом шаге, только вот после 1000 итераций получаются результаты, не имеющие ничего общего с действительностью. Целые тома написаны на тему о том, как избегать подобных сюрпризов. Многие рекомендации актуальны и по сей день, но кое-что удается решить быстро и просто: используйте тип double вместо float, а для промежуточных вычислений не повредит и тип long double.

Например, авторы книги «Writing Scientific Software» рекомендуют избегать однопроходного метода вычисления дисперсии ([Oliveira 2006] стр. 24). Они приводят пример плохо обусловленной задачи. Как вы знаете, число с плавающей точкой называется так, потому что десятичная точка сдвигается в нужную позицию числа, которое во всем остальном не зависит от масштаба. Для простоты изложения допустим, что компьютер работает в десятичной системе счисления; в такой системе число 23 000 000 можно было бы сохранить так же про-

сто, как 0.23 или 0.00023, – достаточно сместить десятичную точку. Но вот число 23 000 000.00023 уже представляет проблему, потому что значащих цифр слишком много (см. пример 7.3).

Пример 7.3 ❖ В числе типа float невозможно сохранить так много значащих цифр (floatfail.c)

```
#include <stdio.h>

int main(){
   printf("%f\n", (float)333334126.98);
   printf("%f\n", (float)333334125.31);
}
```

На моем нетбуке, где тип float занимает 32 разряда, эта программа печатает:

```
333334112.000000
333334112.000000
```

От точности ничего не осталось. Именно поэтому в старых книгах по вычислительной математике так много внимания уделялось алгоритмам, которые минимизируют погрешность, связанную с тем, что надежных значащих цифр всего семь.

Так обстоит дело с 32-разрядным типом float, который на сегодняшний день является самым узким типом с плавающей точкой. Мне даже пришлось явно выполнить приведение к float, потому что иначе система представляла эти числа в виде 64-разрядных значений.

64 разрядов достаточно для надежного хранения 15 значащих цифр: представить число 100 000 000 000 001 больше не проблема. (Попробуйте сами! Подсказка: `printf(%.20g, val)` печатает `val` с 20 значащими десятичными цифрами.)

В примере 7.4 приведена реализация алгоритмов из книги Оливейра и Стюарта, в том числе однопроходный алгоритм вычисления среднего и дисперсии. Повторю, что этот код служит лишь для демонстрации, потому что в библиотеке GSL уже есть готовые калькуляторы среднего и дисперсии. Приведены две версии кода: одна – плохо обусловленная, на которой у авторов в 2006 году получились ужасающие результаты, а вторая – с вычитанием 34 120 из каждого числа, в результате чего получаются значения, для которых даже при использовании типа float удается добиться приемлемой точности. Ну а если входные данные не являются плохо обусловленными, то можно с уверенностью утверждать, что результаты точны.

Пример 7.4 ❖ Плохо обусловленная задача: теперь не проблема (stddev.c)

```
#include <math.h>
#include <stdio.h> //size_t

typedef struct meanvar {double mean, var;} meanvar;

meanvar mean_and_var(const double *data){
  long  double avg = 0,                              ❶
        avg2 = 0;
  long double ratio;
```

```
  size_t cnt= 0;

  for(size_t i=0; !isnan(data[i]); i++){
     ratio = cnt/(cnt+1.0);
     cnt ++;
     avg *= ratio;
     avg2 *= ratio;
     avg += data[i]/(cnt +0.0);
     avg2 += pow(data[i], 2)/(cnt +0.0);
  }
  return (meanvar){.mean = avg,                               ❷
                   .var = avg2 - pow(avg, 2)}; //E[x^2] - E^2[x]
}

int main(){
   double d[] = { 34124.75, 34124.48,
                  34124.90, 34125.31,
                  34125.05, 34124.98, NAN};

   meanvar mv = mean_and_var(d);
   printf("mean:%.10g var:%.10g\n", mv.mean, mv.var*6/5.);
   double d2[] = { 4.75, 4.48,
                   4.90, 5.31,
                   5.05, 4.98, NAN};

   mv = mean_and_var(d2);
   mv.var *= 6./5;                                            ❸
   printf("mean:%.10g var:%.10g\n", mv.mean, mv.var);         ❹
}
```

❶ Вообще говоря, использование более высокой точности для вычисления промежуточных результатов помогает избежать накопления ошибок округления. То есть если конечный результат должен иметь тип `double`, то `avg`, `avg2` и `ratio` следует объявить как `long double`. Изменятся ли результаты в этом примере, если мы всюду будем использовать тип `double`? (Ответ: нет.)

❷ Функция возвращает структуру, созданную с помощью позиционных инициализаторов. Если вы еще незнакомы с этой формой, то скоро познакомитесь.

❸ Функция в предыдущей строке вычислила дисперсию генеральной совокупности; масштабируем для получения дисперсии выборки.

❹ В качестве форматных спецификаторов в `printf` я указал`%g`; такой спецификатор принимает значения обоих типов: `float` и `double`.

Вот какие получились результаты:

```
mean: 34124.91167 var: 0.07901676614
mean: 4.911666667 var: 0.07901666667
```

Средние отклоняются от 34 120, потому что мы так организовали вычисления, но в остальном практически одинаковы (если мы бы позволили, то цифры .66666 так и печатались бы до конца страницы и дальше), и дисперсия при плохо обусловленных данных отклоняется от истинной всего на 0.000125%. Плохая обусловленность не дает заметного эффекта.

Это, любезный читатель, технический прогресс. Стоило нам удвоить объем памяти под задачу, как все трудности испарились. *Да, можно построить реальный пример, когда накопление погрешностей способно создать проблемы*, но сделать это гораздо труднее. И даже если между временем выполнения программ, написанных с использованием `double` и `float`, существует поддающаяся измерению разница, то несколькими лишними микросекундами можно пожертвовать, чтобы избежать других мучений.

Следует ли использовать тип `long int` всюду, где раньше использовались просто целые? Этот вопрос далеко не так очевиден. Представление числа π типом `double` точнее, чем типом `float`, хотя речь в обоих случаях идет о приближениях; но представления чисел, не превышающих двух миллиардов, типами `int` и `long int` в точности идентичны. Единственная проблема – переполнение. Было время, когда предел был шокирующе маленьким – порядка 32 000. Хорошо все-таки жить в наше время, когда диапазон целых чисел в типичной системе составляет примерно ±2,1 миллиарда. Но если имеется хотя бы отдаленная возможность, что в результате умножений переменная окажется больше нескольких миллиардов (а это всего-то $200 \times 200 \times 100 \times 500$), то, безусловно, следует использовать тип `long int` или даже `long long int`, иначе ответ будет не просто неточным, а совершенно неправильным, потому что в большинстве систем переход через границу +2,1 миллиарда приводит к циклическому возврату к величине −2,1 миллиарда. Загляните в файл *limits.h* в своей системе (обычно он находится в каталоге */include* или */usr/include/*); на моем нетбуке этот файл говорит, что типы `int` и `long int` одинаковы.

Если вы занимаетесь серьезными вычислениями, включайте директиву `#include <stdint.h>` и пользуйтесь типом `intmax_t`, верхняя граница которого гарантированно составляет не менее $2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807$ (C99 §7.18.1 и C11 §7.20.1).

Если вы все-таки соберетесь поменять типы, не забудьте во всех вызовах `printf` заменить спецификатор`%i` на`%li` в случае `long int` и на`%ji` в случае `intmax_t`.

Сравнение чисел без знака

Ниже приведена простая программа, в которой `int` сравнивается с `size_t` – беззнаковым целым типом, который иногда используется для представления адресов памяти (формально это тип, который возвращает `sizeof`):

Пример 7.5 ❖ Сравнение целых со знаком и без знака (uint.c)

```
#include <stdio.h>

int main(){
   int neg = -2;
   size_t zero = 0;
   if (neg < zero) printf("Да, -2 меньше 0.\n");
   else            printf("Нет, -2 не меньше 0.\n");
}
```

Запустите программу и убедитесь, что она печатает неправильный ответ. Этот пример показывает, что в большинстве операций сравнения целых со знаком и без знака С принудительно приводит знаковый тип к беззнаковому (C99 и C11

§6.3.1.8(1)), что прямо противоположно интуитивно ожидаемому. Признаюсь, я сам несколько раз нарывался на это, а найти такую ошибку очень трудно, потому что сравнение выглядит абсолютно естественно.

В языке C есть много способов представления числа – от `unsigned short int` до `long double`. Такое изобилие типов было необходимо во времена, когда даже у больших ЭВМ объем памяти измерялся килобайтами. Но сегодня не стоит использовать всё это многообразие. Чрезмерно детальное управление типами, например применение `float` ради повышения эффективности, а `double` – только в особых случаях, или объявление переменной типа `unsigned int`, потому что вы уверены, что отрицательное значение для нее невозможно, – все это открывает двери для тонких ошибок, связанных с неточностью представления чисел и интуитивно неочевидными правилами преобразования типов в C.

Безопасное преобразование строки в число

Существует несколько функций для разбора текстовых строк с целью выделения числа. Самые популярные – `atoi` и `atof` (ASCII-to-int и ASCII-to-float). Работать с ними очень просто:

```
char twelve[] = "12";
int x = atoi(twelve);

char million[] = "1e6";
double m = atof(million);
```

Но никакой проверки ошибок в них нет: если переменная `twelve` содержит строку `"XII"`, то `atoi(twelve)` вернет 0, и программа продолжит работу.

Более безопасны функции `strtol` и `strtod`. Они появились еще в стандарте C89, но остаются не слишком востребованными, потому что не упоминаются в первом издании книги K&R, да и для работы с ними нужно приложить чуть больше усилий. Большинство знакомых мне авторов (в том числе и я сам в предыдущей книге) не упоминают их вовсе или выносят в приложение.

Функция `strtod` принимает второй аргумент, указатель на указатель на `char`, который указывает на первый символ, который анализатор не смог интерпретировать как часть числа. Его можно использовать для разбора оставшейся части текста или для контроля ошибок, если вы ожидаете, что строка должна содержать только число и ничего более. Если объявить эту переменную как `char *end`, то после разбора строки, которая содержит только число, `end` будет указывать на `'\0'` в конце строки, поэтому для диагностики ошибки достаточно написать что-то вроде `if (*end) printf("ошибка чтения")`.

В примере 7.6 приведена простая программа, которая возводит в квадрат число, заданное в командной строке.

Пример 7.6 ❖ Применение strtod для чтения числа (strtod.c)

```
#include "stopif.h"
#include <stdlib.h> //strtod
```

```
#include <math.h> //pow

int main(int argc, char **argv){
   Stopif (argc < 2, return 1, "Задайте в командной строке число, "
                               "которое нужно возвести в квадрат.");
   char *end;
   double in = strtod(argv[1], &end);
   Stopif(*end, return 2, "Ошибка при преобразовании '%s' в число. "
        "Проблема, начиная с '%s'.", argv[1], end);
   printf("Квадрат%s равен%g\n", argv[1], pow(in, 2));
}
```

Начиная со стандарта C99, существуют также функции `strtof` и `strtold` для преобразования строки в `float` и `long double`. Функции `strtol` и `strtoll` для преобразования в `long int` и `long long int`, соответственно, принимают три аргумента: преобразуемую строку, указатель на конец и основание системы счисления. Обычно задается основание 10, но можно указать 2 для чтения двоичных чисел, 8 – для восьмеричных, 16 – для шестнадцатеричных и т. д. вплоть до 36.

Пометка недопустимых числовых значений с помощью маркера NaN

Надо выстоять, выстоять надо,
Деление на нуль надвигается,
как разборщиков бригада.
— The Offspring
«Dividing by zero»

В стандарте IEEE представления чисел с плавающей точкой сформулированы точные правила, в том числе специальные представления положительной и отрицательной бесконечности и нечисла (NaN) – маркера, обозначающего математическую ошибку типа 0/0 или log(−1). IEEE 754/IEC 60559 (так официально называется этот стандарт, потому что люди, занимающиеся подобными вещами, предпочитают, чтобы у стандартов были числовые названия), отличается от стандартов C и POSIX, но поддерживается почти всюду. Если вы работаете на компьютере Cray или на каком-то узкоспециализированном встроенном устройстве, то изложенные в этом разделе подробности можете проигнорировать (но даже в библиотеке AVR libc для Arduino и других микроконтроллеров определены константы `NAN` и `INFINITY`).

Как видно из примера 10.1, маркер NaN может быть полезен для обозначения конца списка при условии, что в самом списке гарантированно нет значений NaN.

Еще одна вещь, которую нужно обязательно знать о NaN, – тот факт, что сравнение с этим значением на равенство *всегда* заканчивается неудачно – после присваивания `x=NAN` даже вычисление выражения `x==x` дает false. Чтобы проверить, совпадает ли `x` с NaN, нужно воспользоваться функцией `isnan(x)`.

Тех, кому приходится постоянно заниматься численными расчетами, могут заинтересовать другие способы использования NaN в качестве маркера.

В стандарте IEEE определено *множество* форм NaN: знаковый бит может быть равен 0 или 1, показатель степени должен содержать только единицы, а все остальное отлично от нуля, то есть получается такая последовательность битов: S11111111MMMMMMMM MMMMMMMMMMMMMMM, где S – знак, а M – неопределенная мантисса.

Нулевая мантисса означает ±бесконечность в зависимости от знакового бита, но в остальном мы свободны в качестве M выбирать все, что угодно. Придумав, как интерпретировать эти биты, мы сможем включать различные признаки в числовые массивы.

Изящный способ сгенерировать конкретное нечисло – воспользоваться функцией `nan(tagp)`, которая возвращает нечисло, «содержимое которого определяется параметром `tagp`». (C99 и C11 §7.12.11.2). Параметр должен быть строкой, представляющей

число с плавающей точкой, – функция nan обертывает функцию strtod – которое будет записано в мантиссу создаваемого нечисла.
Программа в примере 7.7 генерирует и использует маркер NA (нет данных), полезный в контексте, когда нужно различать отсутствующие данные и математические ошибки.

Пример 7.7 ❖ Создание собственного маркера NA для аннотации данных с плавающей точкой (na.c)

```
#include <stdio.h>
#include <math.h> //NAN, isnan, nan

double ref;

double set_na(){
   if (!ref) ref=nan("21");
   return ref;
}

int is_na(double in){          ❶
   if (!ref) return 0; //set_na еще не вызывалась ==> маркеров NA пока нет.
   char *cc = (char *)(&in);
   char *cr = (char *)(&ref);
   for (int i=0; i< sizeof(double); i++)
      if (cc[i] != cr[i]) return 0;
   return 1;
}

int main(){
   double x = set_na();
   double y = x;
   printf("Is x=set_na() NA?%i\n", is_na(x));
   printf("Is x=set_na() NAN?%i\n", isnan(x));
   printf("Is y=x NA?%i\n", is_na(y));
   printf("Is 0/0 NA?%i\n", is_na(0/0.));
   printf("Is 8 NA?%i\n", is_na(8));
}
```

❶ Функция is_na сравнивает комбинацию бит проверяемого числа со специальной комбинацией бит, заданной с помощью set_na. Для этого она рассматривает обе комбинации как символьные строки и производит посимвольное сравнение.

Я создал один признак, записываемый вместо числа, произвольно выбрав в качестве ключа число 21. Немного модифицировав этот код, мы сможем сгенерировать сколько угодно различных маркеров для обозначения различных исключительных ситуаций.
На самом деле некоторые широко распространенные системы (например, WebKit) идут гораздо дальше и вставляют в мантиссу нечисел не просто признак, а настоящий указатель. Реализацию этого метода, который называется *упаковкой в NaN* (NaN boxing), оставляю читателю в качестве упражнения.



Глава 8

Важные особенности синтаксиса C, которые в учебниках часто не рассматриваются

В предыдущей главе рассматривались некоторые вопросы, которым в традиционных учебниках C уделяется много внимания, но которые при работе на современных компьютерах уже не так существенны. В этой главе мы рассмотрим темы, которые, как я обнаружил, во многих учебниках не рассматриваются вовсе или затрагиваются вскользь. Как и выше, обсуждается много мелких вопросов, объединенных в три крупных раздела.

- Препроцессор часто незаслуженно обходят вниманием, наверное, потому что многие думают, будто это какой-то вспомогательный инструмент, а не настоящий C. Однако он включен не без причины: есть вещи, которые в C можно сделать только с помощью макросов. Не все совместимые со стандартами компиляторы предлагают одинаковый набор средств, и именно с помощью препроцессора мы распознаем характеристики окружения и так или иначе реагируем на них.
- Делая обзор учебников C, я наткнулся на одну-две книги, в которых ключевые слова `static` и `extern` даже не упоминаются. Поэтому в этой главе мы потратим некоторое время на обсуждение *компоновки* и рассмотрим плохо понимаемые применения ключевого слова `static`.
- Ключевое слово `const` попало в эту главу, потому что оно настолько полезно, что не использовать его было бы преступлением, но при этом его описание в стандарте и реализация в распространенных компиляторах содержит некоторые странности.

Выращивание устойчивых и плодоносящих макросов

В главе 10 описывается несколько способов сделать интерфейс с библиотекой более дружелюбным к пользователю и менее подверженным ошибкам. Для этой цели активно применяются макросы.

Мне не раз доводилось слышать, будто макросы – сами по себе источник ошибок и потому их следует избегать, но люди, которые так говорят, почему-то не советуют отказаться от `NULL`, `isalpha`, `isfinite`, `assert`, обобщенных функций, адаптирующихся к типу аргументов, в частности `log`, `sin`, `cos` и `pow`, а также десятков других средств, которые в стандартной библиотеке GNU реализованы в виде макросов. И это надежные, правильно написанные макросы, которые в любых обстоятельствах делают то, для чего предназначены.

Макрос производит текстовую подстановку (которая называется *расширением* в предположении, что подставляемый текст длиннее исходного), а к этой операции следует относиться иначе, чем к вызову функции, потому что входной текст может взаимодействовать с текстом в макросе и соседним текстом в исходном коде. Макросы лучше использовать в тех случаях, когда такое взаимодействие желательно и предотвращать его нет необходимости.

Прежде чем переходить к правилам написания надежных макросов – а их всего три, – давайте выделим два типа макросов. В первом случае результатом расширения является выражение, а это значит, что мы можем вычислять такие макросы, печатать их значения или – если результаты числовые – использовать внутри уравнения. Во втором случае расширение дает блок предложений, который может встречаться после `if` или в цикле `while`. А теперь сами правила.

- Скобки! Очень легко получить не то, что ожидаешь, после того как макрос вставит какой-то текст. Вот простой пример:

  ```
  #define double(x) 2*x      Нужны дополнительные скобки.
  ```

  Если теперь пользователь напишет `double(1+1)*8`, то в результате расширения макроса получится `2*1+1*8`, что дает 10, а не 32. Чтобы все было правильно, нужно добавить скобки:

  ```
  #define double(x) (2*(x))
  ```

  Теперь вычисление `(2*(1+1))*8` дает правильный результат. Общее правило состоит в том, чтобы заключать в скобки все входные параметры, если нет особых причин поступить иначе. Для макросов, расширяющихся в выражение, сам результат макроса также нужно заключать в скобки.

- Избегайте двойной подстановки. Следующий пример из учебника несет в себе опасность:

  ```
  #define max(a, b) ((a) > (b) ? (a) : (b))
  ```

  Написав `int x=1, y=2; int m=max(x, y++)`, пользователь ожидает, что `m` будет равно 2 (значение `y` до инкремента), после чего `y` примет значение 3. Но макрос-то расширяется следующим образом:

```
m = ((x) > (y++) ? (x) : (y++))
```

а это означает, что y++ вычисляется дважды, то есть инкремент выполняется два раза, хотя пользователь ожидал однократного, и в результате m будет равно 3, а не 2.

В случае макроса блочного типа мы можем в начале блока объявить переменную, которая принимает значение, равное входному параметру, и в остальной части макроса использовать эту копию.

Этому правилу следуют не так безусловно, как правилу скобок – макрос max часто встречается в неожиданных местах, – поэтому помните, что побочные эффекты внутри неизвестных макросов следует сводить к минимуму.

- Фигурные скобки для блоков. Вот простой блочный макрос:

```
#define doubleincrement(a, b) \ Нужны фигурные скобки.
    (a)++;                    \
    (b)++;
```

Он сделает совершенно не то, что нужно, если поставить его после if:

```
int x=1, y=0;
if (x>y)
   doubleincrement(x, y);
```

Если добавить отступы, чтобы сделать ошибку очевидной, то мы увидим такой результат расширения:

```
int x=1, y=0;
if (x>y)
    (x)++;
(y)++;
```

Еще один подвох: что, если в макросе объявлена переменная total, а пользователь уже объявил такую же переменную в объемлющей программе? Переменные, объявленные в блоке, конфликтуют с одноименными переменными, объявленными вне блока. В примере 8.1 показано простое решение обеих проблем: заключайте макрос в фигурные скобки.

Если весь макрос заключен в фигурные скобки, то мы можем завести промежуточную переменную с именем total, которая будет существовать только в области видимости внутри этих фигурных скобок и никак не испортит переменную total, объявленную в main.

Пример 8.1 ❖ Мы можем управлять областью видимости с помощью фигурных скобок, как и в обычном коде безо всяких макросов (curly.c)

```
#include <stdio.h>

#define sum(max, out) { \
   int total=0;     \
   for (int i=0; i<= max; i++) \
      total += i;  \
   out = total;     \
}

int main(){
```

```
    int out;
    int total = 5;
    sum(5, out);
    printf("out=%i original total=%i\n", out, total);
}
```

Но одна мелкая шероховатость все-таки остается. Вернемся к простому макросу doubleincrement; код

```
#define doubleincrement(a, b) { \
    (a)++;    \
    (b)++;    \
}

if (a>b) doubleincrement(a, b);
else return 0;
```

расширяется в:

```
if (a>b) {
    (a)++;
    (b)++;
};
else return 0;
```

Лишняя точка с запятой перед else смущает компилятор. Пользователь получит сообщение об ошибке компиляции и, стало быть, поставлять заказчику такой код нельзя, однако решение – убрать точку с запятой или заключить предложение в дополнительные фигурные скобки, кажущиеся совершенно излишними, – не очевидно, и интерфейс такого макроса не назовешь прозрачным. Честно говоря, тут мало что можно сделать. Типичное решение – обернуть макрос в однократно исполняемый цикл do-while:

```
#define doubleincrement(a, b) do {  \
    (a)++;  \
    (b)++;  \
} while(0)

if (a>b) doubleincrement(a, b);
else return 0;
```

Это решает проблему, и мы получаем макрос, про который пользователи могут и не знать, что это макрос. Но что, если в макросе имеется предложение break – встроенное или каким-то образом включенное пользователем? Вот очередной пример макроса утверждения и случай, когда он не работает:

```
#define AnAssert(expression, action) do { \
if (!(expression)) action; \
} while(0)

double an_array[100];
```

```
double total=0;
...
for (int i=0; i< 100; i++){
   AnAssert(!(isnan(an_array[i])), break);
   total += an_array[i];
}
```

Пользователь не знает о том, что заданное им предложение break оказывается в цикле do-while, внутреннем для макроса, поэтому откомпилирует и запустит неправильный код. В тех случаях, когда обертка do-while может нарушать ожидаемое поведение break, пожалуй, проще всего убрать обертку и предупредить пользователей о неприятности, касающейся точки с запятой перед else[1].

Выполнив команду gcc -E curly.c, мы увидим, что препроцессор расширяет макрос sum, как показано ниже, и благодаря наличию фигурных скобок переменная total в области видимости макроса никак не влияет на переменную total в области видимости main. Поэтому будет напечатано значение total, равное 5:

```
int main(){
  int out;
  int total = 5;
  { int total=0; for (int i=0; i<= 5; i++) total += i; out = total; };
  printf("out=%i total=%i\n", out, total);
}
```



Ограничение области видимости макроса с помощью фигурных скобок не защищает от всех видов конфликтов имен. Что случится в предыдущем примере, если мы напишем int out, i=5; sum(i, out);?



Если имеется подозрительный макрос, запустите gcc, Clang или icc с флагом -E, в результате отработает только препроцессор, и расширенная версия обрабатываемого файла будет выведена на stdout. Поскольку сюда входит и содержимое #include <stdio.h> и прочих весьма объемных стандартных файлов, то я обычно перенаправляю вывод в файл или в программу постраничного просмотра командой gcc -E mycode.c | less, а уже затем ищу расширения отлаживаемого макроса.

Вот, собственно, и все, что можно сказать о капризах макросов. Основной принцип – стремиться к простоте макросов – сохраняет силу, и, как вы убедитесь, реальные макросы чаще всего умещаются в одну строку, их задача – тем или иным способом подготовить параметры, а затем вызвать стандартную функцию для выполнения реальной работы. Отладчик и сторонние системы не располагают средствами разбора макроопределений, поэтому все написанное вами должно работать и без макросов. В разделе «Компоновка с ключевыми словами static и extern» ниже приводится рекомендация о том, как упростить себе жизнь при написании простых функций.

[1] Существует также возможность обернуть блок в предложение if (1) { … } else (void) 0, которое тоже поглощает точку с запятой. Технически это работает, но приводит к предупреждению компилятора, когда сам макрос вложен в предложение if-else и задан флаг -Wall. Поэтому такое решение тоже не вполне прозрачно для пользователя.

Приемы работы с препроцессором

Знак решетки #, зарезервированный для препроцессора, используется последним тремя совершенно разными способами: для обозначения директив, для преобразования аргументов в строку и для конкатенации лексем.

Как вы знаете, директива препроцессора, например #define, должна начинаться знаком # в начале строки.

Попутно отметим, что пробелы, предшествующие знаку #, игнорируются (K&R, 2-е издание, §A12, стр. 228), и это полезно с точки зрения оформления кода. Например, одноразовые макросы можно поместить внутрь функции, непосредственно перед тем местом, где они используются, с таким же отступом, как в самой функции. Согласно воззрениям старой школы, размещение макроса в месте использования – свидетельство «неправильной» организации программы (а правильно – располагать все макросы в начале файла), но, располагая макросы по месту, мы упрощаем себе ссылку на них и подчеркиваем их одноразовую природу. В разделе «OpenMP» на стр. 293 мы будем аннотировать циклы for директивами #pragma, и размещение знака # в первой позиции строки превратило бы текст программы в нечитаемый хаос.

Следующее использование знака # – преобразование аргумента макроса в строку. Программа в примере 8.2 демонстрирует одну особенность оператора sizeof (см. врезку), но ее основная цель – показать использование макроса препроцессора.

Пример 8.2 ❖ Текст одновременно печатается и вычисляется (sizesof.c)

```
#include <stdio.h>

#define Peval(cmd) printf(#cmd ":%g\n", cmd);

int main(){
   double *plist = (double[]){1, 2, 3};      ❶
   double list[] = {1, 2, 3};
   Peval(sizeof(plist)/(sizeof(double)+0.0));
   Peval(sizeof(list)/(sizeof(double)+0.0));
}
```

❶ Это составной литерал. Если вы с ними незнакомы, подождите немного, я расскажу позже. Глядя на то, как sizeof обрабатывает plist, помните, что plist – это указатель на массив, а не сам массив.

Выполнив эту программу, вы увидите, что аргумент макроса печатается как обычный текст, а затем его значение вычисляется, поскольку #cmd эквивалентно строке "cmd". Таким образом, Peval(list[0]) должно было бы быть расширено в:

```
printf("list[0]" ":%g\n", list[0]);
```

Вас смущают две строки "list[0]" ":%g\n", расположенные вплотную друг к другу? Следующий трюк препроцессора заключается в том, что две соседние литеральные строки объединяются в одну: "list[0]:%g\n". И это справедливо не только по отношению к макросам:

```
printf("Конкатенацию строк препроцессором "
       "можно использовать для разбиения длинных "
       "строк в программе. Мне кажется, что это "
       "проще, чем использовать знак обратной "
       "косой черты, но следите за пробелами.");
```

Ограничения sizeof

Вы попробовали запустить код из предыдущего примера? Он основан на известном трюке, позволяющем получить размер автоматического или статического массива путем деления общего размера массива на размер одного его элемента (**http://c-faq.com/aryptr/arraynels.html**; см. также K&R, 1-е издание, стр. 126; 2-е издание, стр. 135), например:

```
// Это ненадежно:
#define arraysize(list) sizeof(list)/sizeof(list[0])
```

Оператор `sizeof` (это ключевое слово С, а не обычная функция) применяется к автоматически выделенной переменной (которая может быть массивом или указателем), а не к данным, на которые направлен указатель. В случае автоматического массива вида `double list[100]` компилятор должен выделить память для 100 чисел типа `double` и позаботиться о том, чтобы эту область (скорее всего, 800 байт) не затерла следующая переменная, размещенная в стеке. В случае динамически выделенной памяти (`double *plist; plist = malloc(sizeof(double *100));`) указатель в стеке занимает, наверное, 8 байтов (ну уж заведомо не 100), и `sizeof` возвращает длину этого указателя, а не того, на что он указывает.

Когда вы указываете пальцем на игрушку, одни кошки обследуют игрушку, а другие обнюхивают ваш палец.

А бывает, что нужно соединить два объекта, не являющиеся строками. В этом случае используются два знака решетки подряд: `##`. Если переменная `name` принимает значение `LL`, то конструкция `name ## _list` преобразуется в `LL_list` – это допустимое имя переменной, которым можно воспользоваться в программе.

«А вот хотелось бы, – слышу я ваше замечание, – чтобы с каждым массивом была ассоциирована переменная, содержащая его длину». Что ж, в примере 8.3 приведен макрос, который объявляет локальную переменную с именем, оканчивающимся на `_len`, для каждого интересующего вас списка. Он даже позаботится о наличии завершающего маркера у списка, так что и длина-то не нужна.

Впрочем, сам по себе такой макрос – перебор, и я не рекомендую сразу же бросаться применять его, однако он демонстрирует, как можно сгенерировать множество временных переменных с единообразно устроенными именами.

Пример 8.3 ❖ Создание вспомогательных переменных с помощью препроцессора (preprocess.c)

```
#include <stdio.h>
#include <math.h> //NAN

#define Setup_list(name, ...) \
   double *name ## _list = (double [])(__VA_ARGS__, NAN);\    ❶
   int name ## _len = 0;  \
   for (name ## _len =0;  \
       !isnan(name ## _list[name ## _len]); \
```

```
    ) name ## _len ++;

int main(){
   Setup_list(items, 1, 2, 4, 8);                          ❷
   double sum=0;
   for (double *ptr= items_list; !isnan(*ptr); ptr++)      ❸
      sum += *ptr;
   printf("total for items list:%g\n", sum);

   #define Length(in) in ## _len                           ❹

   sum=0;
   Setup_list(next_set, -1, 2.2, 4.8, 0.1);
   for (int i=0; i < Length(next_set); i++) 5
      sum += next_set_list[i];
   printf("total for next set list:%g\n", sum);
}
```

❶ В левой части показано использование ## для порождения имени переменной по шаблону. Правая часть предвосхищает обсуждение макросов с переменным числом аргументов в главе 10.
❷ Порождаются переменные `items_len` и `items_list`.
❸ Цикл с применением `NaN` в качестве маркера.
❹ В некоторых системах разрешается запрашивать у массива его собственную длину, как показано здесь.
❺ Это цикл, в котором используется переменная `next_set_len`, содержащая длину массива.

Попутно сделаем стилистическое замечание: исторически было принято отличать макросы от функций, записывая их имена заглавными буквами. Это служило предостережением о возможных сюрпризах из-за подстановки текста. Мне такая запись напоминает вопль, поэтому я предпочитаю делать заглавной только первую букву имени макроса. Другие вообще не обращают внимания на регистр букв.

Аргументы макроса необязательны

Вот разумный макрос утверждения, который возвращает информацию о том, что утверждение ложно:

```
#define Testclaim(assertion, returnval) if (!(assertion)) \
   {fprintf(stderr, #assertion " failed to be true. \
   Returning " #returnval "\n"); return returnval;}
```

Вот пример его использования:

```
int do_things(){
   int x, y;
   …
   Testclaim(x==y, -1);
   …
   return 0;
}
```

Но что, если функция не возвращает значение? В таком случае второй аргумент можно оставить пустым:

```
void do_other_things(){
   int x, y;
   …
   Testclaim(x==y, );
   …
   return;
}
```

Тогда последняя строка макроса расширяется в `return ;`, что вполне допустимо и соответствует функции, возвращающей `void`[1].

При желании можно даже реализовать значения по умолчанию:

```
#define Blankcheck(a) {int aval = (#a[0]=='\0') ? 2 : (a+0); \
   printf("Как я понимаю, вы имели в виду%i.\n", aval); \
}
// Использование:
Blankcheck(0); // aval будет присвоено значение 0.
Blankcheck( ); // aval будет присвоено значение 2.
```

Проверочные макросы

Устройства, способные исполнять написанные на С программы, весьма разнообразны, среди них и ПК под управлением Linux, и микроконтроллеры Arduino, и холодильники Дженерал Электрик. Программа определяет возможности компилятора и целевой платформы с помощью проверочных макросов, которые можно определить с помощью флага компилятора `-D` или во включаемых директивой `#include` файлах, описывающих локальную функциональность, например `unistd.h` в POSIX-совместимых системах или `windows.h` (и включаемых в него заголовках) в Windows.

Понимая, какие макросы что проверяют, мы можем использовать препроцессор для адаптации к окружению.

GCC и clang выводят список определенных макросов при запуске с флагами `-E -dM` (`-E` – запускать только препроцессор, `-dM` – распечатывать значения макросов). На машине, которой я пользуюсь для написания этой книги, команда

```
echo "" | clang -dM -E -xc -
```

выводит 157 макросов.

Невозможно включить в книгу полный перечень всех макросов проверки возможностей, включая оборудование, версию стандартной библиотеки С и компилятора, но ниже приведены некоторые наиболее употребительные и стабильные макросы с пояснениями. Я выбрал те макросы, которые в наибольшей степени соответствуют теме книги или опрашивают наиболее общие аспекты системы. Имена, начинающиеся с `__STDC_…`, определены в стандарте С.

---

[1] О допустимости пустых аргументов макроса см. C99 и C11§6.10.3(4), где явно разрешены «аргументы, не содержащие ни одной лексемы препроцессора».

Макрос	Назначение
_POSIX_C_SOURCE	Согласована со стандартом IEEE 1003.1 (он же ISO/IEC 9945). Значением обычно является дата редакции
_WINDOWS	Операционная система Windows, в которой есть заголовок windows.h, содержащий все определения
__MACOSX__	Операционная система Mac OS X
__STDC_HOSTED__	Программа компилируется для компьютера, операционная система которого вызывает функцию main
__STDC_IEC_559__	Согласована со стандартом IEEE 754, описывающим представление чисел с плавающей точкой. Впоследствии стандарт получил название ISO/IEC/IEEE 60559. В частности, процессор умеет представлять NaN, INFINITY и -INFINITY
__STDC_VERSION__	Версия стандарта, реализованная компилятором: часто 199409L обозначает C89 (с исправлениями в редакции 1995 года), 199901L – C99, 201112L – C11
__STDC_NO_ATOMICS__	Равно 1, если реализация не поддерживает типа _Atomic и не содержит файла stdatomic.h
__STDC_NO_COMPLEX__	Равно 1, если реализация не поддерживает типа комплексных чисел
__STDC_NO_VLA__	Равно 1, если реализация не поддерживает массивов переменной длины
__STDC_NO_THREADS__	Равно 1, если реализация не поддерживает описанного в стандарте C заголовка threads.h и определенных в нем элементов. В качестве альтернатив можно использовать потоки POSIX, библиотеку OpenMP и т. д.

Одно из важнейших достоинств Autoconf – умение генерировать макросы, описывающие набор имеющихся возможностей. Предположим, что вы пользуетесь Autoconf, что файл config.ac содержит такую строку:

```
AC_CHECK_FUNCS([strcasecmp asprintf])
```

и что в системе, в которой был запущен скрипт ./configure, имеется (описанная в POSIX) функция strcasecmp, но отсутствует (описанная в стандарте GNU/BSD) функция asprintf. Тогда Autoconf создаст заголовок config.h, содержащий такие строки:

```
#define HAVE_STRCASECMP 1
/* #undef HAVE_ASPRINTF */
```

После этого вы можете адаптировать свой код к реалиям с помощью директив препроцессора #ifdef (если определено) или #ifndef (если не определено), например:

```
#include "config.h"

#ifndef HAVE_ASPRINTF
[вставить сюда исходный код asprintf, см. пример 9.3]
#endif
```

Бывает и так, что без определенной функциональности программа просто не может работать, и тогда можно воспользоваться директивой препроцессора #error:

```
#ifndef HAVE_ASPRINTF
   #error "HAVE_ASPRINTF не определена. Ну не буду я " \
   "компилироваться в системе без asprintf."
#endif
```

Начиная со стандарта C11, определено ключевое слово _Static_assert. Статическое утверждение принимает два аргумента: подлежащее проверке статическое выражение и сообщение пользователю. В совместимом с C11 заголовке assert.h определен не столь уродливо выглядящий макрос static_assert, который расширяется в _Static_assert (C11 §7.2(3)). Например:

```
    #include <limits.h> //INT_MAX
    #include <assert.h>

    _Static_assert(INT_MAX < 33000L, "Ваш компилятор определяет слишком короткий тип int.");

    #ifndef HAVE_ASPRINTF
    static_assert(0, "HAVE_ASPRINTF не определена. Не буду я "
                     "компилироваться в системе без asprintf.");
#endif
```

Буква L в конце 33000L и некоторых комбинаций года и месяца означает, что эти числа следует рассматривать как long int – на случай, если используется компилятор, в котором такие большие целые числа вызывают переполнение int.

Такая форма удобнее, чем #if/#error/#endif, но поскольку она появилась лишь в стандарте, опубликованном в декабре 2011 года, то сама по себе является не вполне переносимой. Например, авторы Visual Studio реализовали макрос _STATIC_ASSERT всего с одним аргументом (само утверждение) и не поддерживают стандартного ключевого слова _Static_assert[1].

Кроме того, комбинация #ifdef/#error/#endif и ключевое слово _Static_assert в значительной степени эквивалентны: согласно стандарту C, оба проверяют *константные-выражения* и печатают *строковый-литерал*, хотя первая делает это на этапе препроцессирования, а вторая – на этапе компиляции (C99 §6.10.1(2) и C11 §6.10.1(3); C11 §6.7.10). Таким образом, на данный момент для проверки отсутствующей функциональности, пожалуй, безопаснее придерживаться директив препроцессора.

Защита заголовков

Что произойдет, если включить в файл два одинаковых typedef'а для одной и той же структуры? Например, что, если в заголовке header.h будут такие определения:

```
typedef struct {
    int a;
    double b;
} ab_s;

typedef struct {
    int a;
    double b;
} ab_s;
```

Человек легко убедится, что эти структуры одинаковы, но компилятор обязан считать любое новое объявление структуры в файле новым типом (C99 §6.7.2.1(7)

[1] http://msdn.microsoft.com/en-us/library/bb918086.aspx.

и C11 §6.7.2.1(8)). Поэтому приведенный выше код не откомпилируется, так как символ ab_s объявлен дважды и обозначает два разных (хотя и совпадающих) типа[1].

Получить такую же ошибку из-за двойного объявления можно, если определить typedef только один раз, но включить содержащий его заголовок дважды:

```
#include "header.h"
#include "header.h"
```

Поскольку включаемые файлы зачастую включают другие файлы, а те – третьи и т. д., то такая ошибка может проявиться неожиданным образом из-за длинных цепочек включения. Совместимое со стандартом C решение, предотвращающее такое развитие событий, обычно называется *защитой включения* (include guard) и сводится к определению ассоциированного с файлом символа, наличие которого проверяется в директиве #ifndef:

```
#ifndef Already_included_head_h
#define Already_included_head_h 1
[поместить сюда все содержимое header.h]
#endif
```

При первом включении файла символ еще не определен и содержимое файла разбирается; при втором включении символ определен, поэтому содержимое файла пропускается.

Этот прием используется с незапамятных времен (см. K&R, 2-е издание, §4.11.3), но с появлением прагмы once он стал немного проще. В начало файла, который должен включаться только один раз, помещается строка:

```
#pragma once
```

и компилятор понимает, что второй раз включать файл не нужно. Прагмы – механизм, специфичный для каждого компилятора, и в стандарте C определены лишь немногие. Однако прагму #pragma once поддерживают все основные компиляторы, включая gcc, clang, Intel, Visual Studio в режиме C89 и ряд других.

Комментирование кода средствами препроцессора

Блок, помещенный между директивами #if 0 и #endif, игнорируется, и, значит, этот прием можно использовать, чтобы закомментировать большой участок кода. В отличие от /* ... */, такие комментарии могут быть вложенными:

```
#if 0
...
   #if 0
       /* код, который уже игнорируется */
   #endif
...
#endif
```

[1] Если типы одинаковы, то повторяющиеся typedef'ы – не проблема, потому что в соответствии с C11 §6.7(3) «имя псевдонима типа может быть повторно определено для обозначения того же типа, которое уже обозначает, при условии что этот тип не является модифицируемым переменной».

Но если вложенность некорректна, например:

```
#if 0
...
   #ifdef This_line_has_no_matching_endif
...
#endif
```

то возникнет ошибка, потому что препроцессор сопоставит `#endif` не с тем `#if`.

Компоновка с ключевыми словами static и extern

В этом разделе мы напишем код, который будет сообщать компилятору, что тот должен сказать компоновщику. Компилятор обрабатывает по одному *c*-файлу за раз и (обычно) порождает один *o*-файл. После этого компоновщик связывает все *o*-файлы вместе, создавая библиотеку или исполняемый файл.

Что случится, если в двух разных файлах встретятся объявления одной и той же переменной *x*? Быть может, автор одного файла не знал, что автор другого уже выбрал имя `x`, так что на самом деле эти две `x` должны были бы находиться в разных пространствах имен? А быть может, оба автора прекрасно знали, что ссылаются на одну и ту же переменную, и компоновщик должен сделать так, чтобы все ссылки на `x` вели на одну и ту же область памяти.

Под *внешней компоновкой* понимается, что одинаковые символы, определенные в разных файлах, должны рассматриваться компоновщиком как один символ. Для обозначения внешней компоновки применяется ключевое слово `extern` (см. ниже)[1].

Говоря о *внутренней компоновке*, имеют в виду, что экземпляр переменной `x` или функции `f()` в некотором файле принадлежит этому файлу и может быть отождествлен только с другими экземплярами `x` или `f()` в той же области видимости (что для объектов, объявленных вне функции, означает область видимость файла). Для обозначения внешней компоновки применяется ключевое слово `static`.

Любопытно, что для внешней компоновки есть ключевое слово `extern`, а для внутренней применяется не `intern`, чего следовало бы ожидать, а `static`. В разделе «Автоматическая, статическая и динамическая память» выше я рассказывал о трех видах памяти. Использование одного и того же слова `static` для описания компоновки и модели памяти объединяет эти концепции, которые когда-то по техническим причинам действительно в какой-то мере перекрывались, но теперь совершенно независимы.

- Для переменных в области видимости файла слово `static` влияет только на компоновку:
  - по умолчанию подразумевается внешняя компоновка, поэтому для изменения ее на внутреннюю нужно добавить ключевое слово `static`;

---

[1] Это идет от стандарта C99 и C11 §6.2.3, где на самом деле говорится о разрешении символов в разных областях видимости, а не только в разных файлах. Но безумные трюки с компоновкой через границы областей видимости в пределах одного файла обычно не применяются.

  - для любой переменной в области видимости файла применяется статическая модель памяти вне зависимости от того, написано static int x, extern int x или просто int x.
- Для переменных в области видимости блока слово static влияет только на модель памяти:
  - по умолчанию подразумевается внутренняя компоновка, поэтому ключевое слово static на компоновку не влияет. Компоновку можно изменить, добавив в объявление переменной слово extern, но так делают редко;
  - по умолчанию модель памяти автоматическая, а наличие ключевого слова static изменяет ее на статическую.
- Для функций слово static влияет только на компоновку:
  - функции определяются только в области видимости файла (хотя gcc поддерживает вложенные функции как расширение). Как и в случае переменных в области видимости файла, по умолчанию подразумевается внешняя компоновка, а наличие ключевого слова static меняет ее на внутреннюю;
  - путаницы с моделями памяти не возникает, потому что функции всегда статические, как и переменные в области видимости файла.

Обычно, чтобы сделать функцию доступной в нескольких *c*-файлах, ее объявление помещают в *h*-файл, а определение – в какой-нибудь один *c*-файл (где у нее по умолчанию будет внешняя компоновка). Это разумное соглашение, которого стоит придерживаться, но довольно часто авторы хотят поместить служебные функции из одной-двух строк (типа max или min) в *h*-файл, который включается всюду. Так можно сделать, предварив объявление функции ключевым словом static, например:

```
// В файле common_fns.h:
static long double max(long double a, long double b){
    (a > b) ? a : b;
}
```

Если директива #include "common_fns.h" встречается в десятке файлов, то в каждом из них компилятор создаст отдельный экземпляр функции max. Но поскольку у этой функции внутренняя компоновка, ни один из файлов не сделает имя max видимым извне, так что все эти экземпляры останутся независимыми и не будут конфликтовать друг с другом. Такое повторное объявление немного увеличивает размер исполняемого файла и время компиляции, но в типичном современном окружении это несущественно.

Переменные с внешней компоновкой в файлах-заголовках

С ключевым словом extern возникает меньше недоразумений, чем со static, потому что оно относится только к компоновке, а не к модели памяти. Типичный способ использования переменной с внешней компоновкой выглядит так.

- В заголовке, который будет включаться всюду, где переменная используется, объявляем переменную с ключевым словом extern, например extern int x.

- Ровно в одном *c*-файле объявляем переменную как обычно, с необязательным инициализатором, например `int x=3`. Как всегда для переменных со статической моделью памяти, если опустить начальное значение (написав просто `int x`), переменная будет инициализирована значением 0 или `NULL`.

И это все, что необходимо сделать, чтобы переменная имела внешнюю компоновку.

Может возникнуть искушение поместить объявление со словом `extern` не в заголовок, а прямо в файл с кодом. Допустим, в файле *file1.c* вы объявили переменную `int x`, затем поняли, что к ней нужен доступ из *file2.c*, поэтому, чтобы долго не возиться, добавили строку `extern int x` в начало этого файла. Это сработает – сегодня. Но через месяц вы изменили объявление в *file1.c* на `double x`, и механизм проверки типов компилятора не будет иметь никаких возражений против *file2.c*. Компоновщик, оставаясь в блаженном неведении, адресует функцию в *file2.c* в область памяти, где хранится переменная `x` типа `double`, и функция радостно прочтет это значение как `int`. Катастрофа! Но ее можно избежать, поместив все объявления с ключевым словом `extern` в заголовок, который включается в оба файла – *file1.c* и *file2.c*. Если хотя бы в одном из них тип изменится, то компилятор сможет обнаружить рассогласование.

Под капотом система должна многое сделать, чтобы можно было объявлять одну и ту же переменную несколько раз, а выделять под нее память однократно. Формально объявление с ключевым словом `extern` считается объявлением (предложением с информацией о типе, которую компилятор проверяет для обеспечения гарантированной согласованности), а не определением (инструкцией по выделению и инициализации памяти). Однако объявление без ключевого слова `extern` считается *предварительным определением*: если компилятор дойдет до конца единицы компиляции (см. ниже) и не увидит определения, то предварительное определение становится настоящим, с обычной инициализацией нулем или `NULL`. Стандарт определяет *единицу компиляции* как один файл после подстановки содержимого всех включаемых файлов (см. C99 и C11 §6.9.2(2)).

Такие компиляторы, как gcc и Clang, обычно понимают под *единицей* всю программу, считая, что программа, в которой есть несколько объявлений без `extern` и нет определений, сворачивает все предварительные определения в единственное настоящее. Даже при задании флага `--pedantic` gcc не обращает внимания на то, используется ключевое слово `extern` или опущено. На практике это означает, что слово `extern` в значительной степени факультативно: компилятор интерпретирует десяток объявлений вида `int x=3` как одно объявление одной переменной с внешней компоновкой. Строго говоря, это противоречит стандарту, но в книге K&R (2-е издание, стр. 227] такое поведение описывается как «обычное в системах UNIX и рассматриваемое как общепринятое расширение стандарта [ANSI]». В книге [Harbison 1991] §4.8 документированы четыре различные интерпретации правил, касающихся `extern`.

Это означает, что если вы хотели, чтобы две одноименные переменные в двух разных файлах считались различными, но не добавили ключевое слово `static`, то

компилятор может объединить обе переменные в одну переменную с внешней компоновкой, что повлечет за собой тонкие и трудноуловимые ошибки. Поэтому не забывайте включать static в объявления переменных с областью видимости файла, которые должны иметь внутреннюю компоновку.

Ключевое слово const

Ключевое слово const невероятно полезно, но связанные с ним правила таят в себе ряд неожиданностей и несогласованностей. В этом разделе мы расскажем о них, чтобы устранить все сюрпризы и облегчить использование const в тех случаях, когда этого требуют правила хорошего тона.

Еще только начиная программировать, мы узнаем, что функциям передаются *копии* входных параметров, однако функция тем не менее может изменить входные данные, если передать ей *копию указателя* на эти данные. Видя, что на вход подается не указатель, мы сразу можем сказать, что оригинальная переменная не изменится. Если же мы видим указатель, то ситуация неопределенная. Списки и строки по сути своей являются указателями, так что, возможно, переданный указатель – знак того, что данные могут модифицироваться, а возможно – просто строка.

Ключевое слово const позволяет автору сделать код более понятным. Этот *квалификатор типа* показывает, что данные, на которые ведет параметр-указатель, не будут изменены внутри функции. Знать о том, что данные не изменятся, весьма полезно, поэтому всюду, где необходимо, пользуйтесь этим ключевым словом.

А вот и первый подвох: компилятор не защищает данные, на которые ведет указатель, от любых модификаций. Данные, помеченные квалификатором const под одним именем, можно модифицировать по другому имени. В примере 8.4 переменные a и b указывают на одни и те же данные, но поскольку у a в заголовке функции set_elmt нет квалификатора const, то с ее помощью можно изменить любой элемент массива b (см. рис. 8.1).

Пример 8.4 ❖ Данные, помеченные квалификатором const под одним именем, можно модифицировать с помощью другого имени (constchange.c)

```
void set_elmt(int *a, int const *b){
   a[0] = 3;
}

int main(){
   int a[10] = {};          ❶
   int const *b = a;
   set_elmt(a, b);
}                           ❷
```

❶ Инициализировать массив нулями.

❷ Эта программа ничего не делает, важно лишь, чтобы она откомпилировалась и выполнилась без ошибок. Если вы хотите убедиться, что элемент b[0] действительно изменился, запустите ее под отладчиком, остановитесь на последней строке и напечатайте значение b.

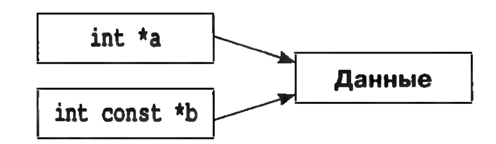


Рис. 8.1 ❖ Мы можем модифицировать данные через a, хотя b имеет квалификатор const; это допустимо

Итак, const – это лишь помощь читателю, а не реальный замок, повешенный на данные.

Форма существительное–прилагательное

Объявления следует читать справа налево. Таким образом:

- int const – константное целое;
- int const * – (неконстантный) указатель на константное целое;
- int * const – константный указатель на (неконстантное) целое;
- int * const * – указатель на константный указатель на целое;
- int const * * – указатель на указатель на константное целое;
- int const * const * – указатель на константный указатель на константное целое.

Как видите, квалификатор const всегда относится к тому, что находится слева от него, – как и *.

Можно переставить местами имя типа и const, то есть int const и const int – одно и то же (хотя проделать этот фокус с const и * не удастся). Я предпочитаю форму int const, потому что она согласуется с более сложными конструкциями и правилом чтения справа налево. Но чаще встречается форма const int, быть может, потому что ее проще произнести на обычном языке (константное целое) или потому что так «всегда делали». Так или иначе, годятся оба варианта.

Как насчет restrict и inline?

Я написал несколько тестов, в которых ключевые слова restrict и inline используются и не используются, – просто чтобы продемонстрировать разницу в быстродействии. Я очень надеялся, что использование restrict в числовых расчетах сможет дать реальный выигрыш, и прошлый опыт это подтверждал. Но, написав недавно эти тесты, я обнаружил, что разница во времени работы пренебрежимо мала.

Следуя собственным рекомендациям, я задавал флаги компиляции CFLAGS=-g -Wall -O3, а это означает, что gcc применял к моим тестовым программам все известные ему приемы оптимизации, и, в частности, понимал, когда трактовать указатели так, будто они объявлены с квалификатором restrict, и когда встраивать функции. А мои указания были ему не нужны.

Конфликты

На практике иногда встречаются ситуации, когда использование const создает конфликты, требующие разрешения. Так бывает, если вы объявили некий указатель константным, но должны передать его функции, в которой соответствующий параметр не помечен квалификатором const. Быть может, автор функции считал,

что это ключевое слово несет с собой слишком много проблем, или поверил болтовне о том, что чем код короче, тем он лучше, а может, просто забыл.

Прежде чем что-то делать, спросите себя, может ли случиться так, что указатель изменится в вызываемой функции без const. Возможно, имеется редкий случай, когда что-то изменяется, или есть еще какая-то странная причина. Об этом в любом случае следует знать.

Если вы убедились, что функция не нарушает обещания константности вашего указателя, то можно без опаски смошенничать и привести константный указатель к типу неконстантного, чтобы заставить компилятор замолчать.

```
// В заголовке этой функции нет const ...
void set_elmt(int *a, int *b){
   a[0] = 3;
}

int main(){
   int a[10];
   int const *b = a;
   set_elmt(a, (int*)b); //...поэтому при вызове приводим тип
}
```

Мне это правило кажется разумным. Мы можем отменить проверку константности, осуществляемую компилятором, при условии что явно заявим о своих намерениях.

Если вы не уверены, что вызываемая функция с уважением отнесется к данному вами обещанию константности, то можете пойти дальше и сделать копию данных, не ограничившись созданием псевдонима. Поскольку вы в любом случае не собирались изменять данных, то после возврата из функции копию можно будет выбросить.

Глубина

Допустим, имеется структура – назовем ее counter_s – и функция, которая принимает указатель на эту структуру: f(counter_s const *in). Может ли такая функция изменять элементы структуры?

Давайте проверим. В примере 8.5 создана структура, содержащая два указателя. Константный указатель на эту структуру передается функции ratio, а затем один из хранящихся в ней указателей мы передаем другой функции, в которой параметр не объявлен как const, и компилятор при этом не ругается.

Пример 8.5 ❖ Элементы константной структуры не являются константными (conststruct.c)

```
#include <assert.h>
#include <stdlib.h> //assert

typedef struct {
   int *counter1, *counter2;
} counter_s;

void check_counter(int *ctr){ assert(*ctr !=0); }
```

```
double ratio(counter_s const *in){                    ❶
   check_counter(in->counter2);
   return *in->counter1/(*in->counter2+0.0);          ❷
}

int main(){
   counter_s cc = {.counter1=malloc(sizeof(int)),     ❸
                   .counter2=malloc(sizeof(int))};
   *cc.counter1 = *cc.counter2 = 1;
   ratio(&cc);
}
```

❶ Входная структура помечена квалификатором const.
❷ Передаем элемент константной структуры функции, принимающей неконстантный параметр. Компилятор не ругается.
❸ Здесь используются позиционные инициализаторы – о них чуть позже.

В определении структуры можно снабдить элемент квалификатором const, хотя обычно это лишнее. Если действительно необходимо защитить только самый нижний уровень в иерархии типов, то лучше оговорить это в документации.

Проблема char const **

В примере 8.6 приведена простая программа, которая проверяет, задано ли в командной строке имя пользователя Iggy Pop. Вот как эта программа вызывается из оболочки (напомним, что переменная $? содержит значение, возвращенное последней запущенной программой):

```
iggy_pop_detector Iggy Pop; echo $?       # печатается 1
iggy_pop_detector Chaim Weitz; echo $?    # печатается 0
```

Пример 8.6 ❖ Неоднозначные формулировки в стандарте вызывают разнообразные проблемы при интерпретации указателя на указатель на константу (iggy_pop_detector.c)

```
#include <stdbool.h>
#include <strings.h> //strcasecmp (из POSIX)

bool check_name(char const **in){           ❶
   return (!strcasecmp(in[0], "Iggy") && !strcasecmp(in[1], "Pop"))
         ||(!strcasecmp(in[0], "James") && !strcasecmp(in[1], "Osterberg"));
}

int main(int argc, char **argv){
   if (argc < 2) return 0;
   return check_name(&argv[1]);
}
```

❶ Если вы раньше не сталкивались с булевым типом, читайте врезку ниже.

Функция check_name принимает указатель на строку, он константный, потому что изменять входные строки нет необходимости. Однако компилятор выдает

предупреждение. В случае Clang оно звучит так: «passing char ** to parameter of type const char ** discards qualifiers in nested pointer types» (передача char ** в качестве параметра типа const char ** приводит к игнорированию квалификаторов во вложенных указательных типах). Имея дело с последовательностью указателей, все компиляторы, к которым у меня был доступ, преобразуют в const то, что можно было бы назвать указателем верхнего уровня (приводят к типу char * const *), но ругаются, если попросить их сделать константным объект, на который указатель указывает (char const **, он же const char **).

Придется выполнить приведение типа явно – заменить check_name(&argv[1]) на:

```
check_name((char const**)&argv[1]);
```

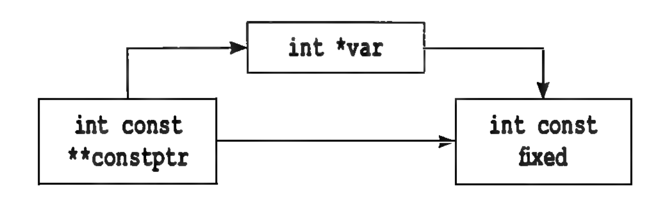
Почему это вполне разумное приведение нельзя было выполнить автоматически? Чтобы увидеть, в чем проблема, придется создать нетривиальные условия, а вся история не согласуется с изложенными ранее правилами. Поэтому объяснение будет непростым, и я не стану осуждать тех, кто захочет его пропустить.

В примере 8.7 создаются три ссылки, показанные на диаграмме: прямая ссылка constptr -> fixed и непрямая ссылка, состоящая из двух шагов: constptr -> var и var -> fixed. Два присваивания в коде производятся явно: constptr -> var и constptr -> -> fixed. Но поскольку *constptr == var, то второе присваивание неявно создает ссылку var -> fixed. Когда мы выполняем присваивание *var=30, переменная fixed получает значение 30.

Пример 8.7 ❖ Мы можем модифицировать данные по альтернативному имени, хотя по основному имени они константны – это должно быть недопустимо (constfusion.c)

```
#include <stdio.h>

int main(){
   int *var;
   int const **constptr = &var; // эта строка - предпосылка ошибки
   int const fixed = 20;
   *constptr = &fixed;           // абсолютно законно
   *var = 30;
   printf("x=%i y=%i\n", fixed, *var);
}
```



Мы никогда не смогли бы разрешить переменной int *var прямо указывать на int const fixed. Это удалось сделать только с помощью хитрой уловки – var неявно указывает на fixed, а открыто об этом нигде не сказано.



Ваша очередь. Можно ли вызвать такую же ошибку с const, когда запрещенное приведение типа происходит в результате вызова функции, как в показанной выше программе распознавания Игги Попа?

Как и раньше, данные, помеченные квалификатором const под одним именем, удается изменить с помощью другого имени. Поэтому не должен вызывать удивления тот факт, что мы сумели изменить константные данные, воспользовавшись альтернативным именем[1].

Я перечисляю проблемы, касающиеся const, чтобы вы знали, как их преодолеть. Но в реальности все не так ужасно, и совет включать const в объявления функций всюду, где это уместно, остается в силе – только не ворчите, что ваши предшественники писали заголовки неправильно. Ведь рано или поздно кто-то воспользуется вашим кодом, и вряд ли вам понравится их ворчание – мол, мы не можем воспользоваться ключевым словом const, потому что он неправильно объявил функции.

Истина и ложь

Изначально в С не было булева типа, принимающего значения true и false, а вместо него действовало соглашение: все, что равно нулю или NULL, – ложь, все остальное – истина. Таким образом, условия if(ptr!=NULL) и if(ptr) эквивалентны.

В стандарте C99 был введен тип _Bool, хотя технически он необязателен, так как всегда можно представить значения true и false целым числом. Но для человека, читающего код, булев тип является прямым указанием на то, что переменная может принимать только значения true или false, то есть проясняет намерения автора.

Комитет по стандартизации остановился на имени _Bool, поскольку оно попадает в пространство имен, зарезервированных для самого языка, но, конечно, выглядит оно ужасно. В заголовке *stdbool.h* определены три макроса для удобства чтения: bool расширяется в _Bool, поэтому вам не придется использовать в объявлениях такой неприглядный подчерк; true расширяется в 1, а false – в 0.

Как и тип bool, макросы true и false проясняют природу присваивания: если я забуду объявить outcome как bool, то выражение outcome=true напомнит о моем намерении, а outcome=1 – нет.

Однако нет никаких фундаментальных причин сравнивать выражение с true или false: все мы привыкли, что if (x) означает если x истинно, то..., и явное добавление части ==true излишне. Кроме того, если в программе есть x=2, то if (x) делает то, что все ожидают, тогда как if (x==true) – нет.

---

[1] Приведенный здесь код заимствован из примера в стандартах C99 и C11 §6.5.16.1(6), в котором строка, аналогичная constptr=&var, названа нарушением ограничения. Почему же gcc и Clang выдают в этом случае предупреждение, но не прекращают компиляцию? Потому что технически все правильно: в разделе C99 и C11 §6.3.2.3(2), относящемся к таким квалификаторам типа, как const, приводится объяснение: «Для любого квалификатора q указатель на тип без квалификатора q может быть преобразован в указатель на тот же тип с квалификатором q...».

Глава 9

Текст

Я верю в то, что слово в итоге разрушит бетон.

— Pussy Riot, цитата из Солженицына в речи от 8 августа 2012 г.

Составленная из букв строка – это массив неопределенной длины, а размер автоматически выделенного в стеке массива нельзя изменить, и в этом корень всех проблем с текстом в C. По счастью, многие до нас уже сталкивались с этой проблемой и предложили решения – пусть даже неполные. Горстки определенных в стандартах C и POSIX функций достаточно для удовлетворения многих потребностей, возникающих при работе со строками.

Кроме того, язык C проектировался в 1970-х годах, когда вопрос о языках, отличных от английского, еще не ставился. Но при использовании правильных функций (и правильного понимания принципов кодирования языка) изначальная приверженность C английскому перестает быть проблемой.

Безболезненная обработка строк с помощью asprintf

Функция `asprintf` выделяет для строки столько памяти, сколько нужно, а затем заполняет ее. Это означает, что вам думать о выделении памяти для строк больше не нужно.

`asprintf` не описана в стандарте C, но имеется в системах, где установлена стандартная библиотека GNU или BSD, а это подавляющее большинство пользователей. К тому же библиотека GNU Libiberty содержит версию `asprintf`, которую можно либо скопировать в свой код, либо вызывать из библиотеки, добавив флаг компоновщика `-liberty`. Библиотека Libiberty входит в состав дистрибутива некоторых систем без встроенной поддержки `asprintf`, в частности MSYS для Windows. А если копирование исходного кода из `libiberty` вас почему-то не устраивает, то ниже приведена простая реализация на базе стандартной функции `vsnprintf`.

Старый способ работы провоцировал программистов на убийство (или на самоубийство – в зависимости от темперамента), поскольку нужно было сначала вычислить длину новой строки, затем выделить для нее память и только потом записать в эту память данные. И не забыть о завершающем нуле!

В примере 9.1 показан трудоемкий способ подготовки строки для передачи функции system, запускающей внешнюю программу. В качестве такой программы взята близкая к рассматриваемой теме утилита strings, которая ищет в двоичном файле печатаемые строки. Функции get_strings передается argv[0], имя самой программы, то есть мы ищем строки в исполняемом файле своей же программы. Это забавно, а чего еще требовать от демонстрационного примера?

Пример 9.1 ❖ Утомительный способ подготовки строк (sadstrings.c)

```
#include <stdio.h>
#include <string.h> //strlen
#include <stdlib.h> //malloc, free, system

void get_strings(char const *in){
   char *cmd;
   int len = strlen("strings ") + strlen(in) + 1;     ❶
   cmd = malloc(len);                                  ❷
   snprintf(cmd, len, "strings%s", in);
   if (system(cmd)) fprintf(stderr, "что-то не так с запуском%s.\n", cmd);
   free(cmd);
}

int main(int argc, char **argv){
   get_strings(argv[0]);
}
```

❶ Предварительное вычисление длины – такая трата времени!
❷ В стандарте C говорится, что sizeof(char)==1, поэтому хотя бы не нужно писать malloc(len*sizeof(char)).

В примере 9.2 используется asprintf, которая сама вызывает malloc, а это означает, что и вычисление длины строки можно опустить.

Пример 9.2 ❖ Эта версия всего на две строки короче приведенной в примере 9.1, но именно эти две строки – источник всех печалей (getstrings.c)

```
#define _GNU_SOURCE //чтобы stdio.h включал asprintf
#include <stdio.h>
#include <stdlib.h> //free

void get_strings(char const *in){
   char *cmd;
   asprintf(&cmd, "strings%s", in);
   if (system(cmd)) fprintf(stderr, "что-то не так с запуском%s.\n", cmd);
   free(cmd);
}

int main(int argc, char **argv){
   get_strings(argv[0]);
}
```

Вызов asprintf очень похож на вызов sprintf, только вместо самой строки нужно передать ее адрес, потому что область памяти для строки будет выделена с помощью malloc, а адрес этой области записан в переданный параметр типа char **.

Но допустим, что по какой-то причине GNU asprintf вам недоступна. Вычисление длины области, необходимой для размещения строки и ее аргументов, – задача, при решении которой легко ошибиться. Как бы заставить компьютер сделать это за нас? За ответом далеко ходить не надо, потому что в стандартах C99 §7.19.6.12(3) и C11 §7.21.6.12(3) написано: «Функция vsnprintf возвращает количество символов, которое было бы записано, если бы n было достаточно велико, без учета завершающего нулевого символа, или отрицательное значение в случае ошибки кодировки». Функция snprintf также возвращает значение, «которое было бы...».

Таким образом, если выполнить пробный вызов vsnprintf, передав ей однобайтовую строку, то в ответ мы узнаем, сколько места нужно выделить для новой строки. После этого мы выделяем память и вызываем vsnprintf по-настоящему. Функция работает дважды, что приводит к удвоению времени, но ради безопасности и удобства на это можно пойти.

В примере 9.3 показана реализация asprintf с помощью описанного подхода: двукратного вызова vsnprintf. Я обернул ее проверкой символа HAVE_ASPRINTF, чтобы можно было учесть результаты Autoconf; см. ниже.

Пример 9.3 ❖ Альтернативная реализация asprintf (asprintf.c)

```
#ifndef HAVE_ASPRINTF
#define HAVE_ASPRINTF
#include <stdio.h> //vsnprintf
#include <stdlib.h> //malloc
#include <stdarg.h> //va_start et al

/* Объявление, нужно поместить в h-файл. Спецификатор __attribute__ сообщает
компилятору, что нужно проверять совместимость типов в духе printf. Это
расширение стандарта C, но большинство компиляторов его поддерживают;
если ваш не относится к их числу, просто уберите этот спецификатор */
int asprintf(char **str, char* fmt, ...) __attribute__ ((format (printf,2,3)));

int asprintf(char **str, char* fmt, ...){
   va_list argp;
   va_start(argp, fmt);
   char one_char[1];
   int len = vsnprintf(one_char, 1, fmt, argp);
   if (len < 1){
      fprintf(stderr, "Ошибка кодировки. Входной указатель сброшен в NULL.\n");
      *str = NULL;
      return len;
   }
   va_end(argp);

   *str = malloc(len+1);
   if (!str) {
      fprintf(stderr, "Не удалось выделить%i байтов.\n", len+1);
      return -1;
   }
   va_start(argp, fmt);
   vsnprintf(*str, len+1, fmt, argp);
```

```
    va_end(argp);
    return len;
}
#endif

#ifdef Test_asprintf
int main(){
    char *s;
    asprintf(&s, "hello,%s.", "-Reader-");
    printf("%s\n", s);

    asprintf(&s, "%c", '\0');
    printf("пустая строка: [%s]\n", s);

    int i = 0;
    asprintf(&s, "%i", i++);
    printf("Нуль:%s\n", s);
}
#endif
```

Безопасность

Если имеется строка фиксированной длины str и вы записываете в нее данные неизвестной длины с помощью sprintf, то можете случайно затереть область по соседству с str – классическое нарушение безопасности. Поэтому вместо sprintf рекомендуется применять функцию snprintf, которая ограничивает длину записываемых данных.

Использование asprintf устраняет эту проблему, потому что выделяется столько памяти, чтобы все записываемые данные в нее гарантированно поместились. Это не дает полной гарантии: конечно, даже в специально подобранной со злым умыслом строке рано или поздно встретится \0, но ее длина может превысить объем свободной памяти, или в str могут быть записаны дополнительные данные, содержащие секретные сведения, например пароль.

В случае нехватки памяти asprintf вернет -1, поэтому если входные данные поступают от пользователя, осторожный программист воспользуется чем-то вроде макроса Stopif (о котором я расскажу ниже в разделе «Макросы с переменным числом аргументов»):

```
Stopif(asprintf(&str, "%s", user_input)==-1, return -1, "asprintf failed.")
```

Впрочем, если вы зашли так далеко, что передаете asprintf непроверенную строку, то терять уже нечего. Заранее проверяйте, что строки из не заслуживающих доверия источников имеют разумную длину. При этом функция может вернуть ошибку, даже когда длина строки не слишком велика, потому что память у компьютера закончилась или ее пожирают злые гремлины.

В стандарте C11 (приложение K) также описаны все обычные функции форматирования с суффиксом _s: printf_s, snprintf_s, fprintf_s и т. д. Предполагается, что они безопаснее функций без суффикса _s. Входная строка не может быть равна NULL, и при попытке записать в строку более RINT_MAX байтов (где RINT_MAX – по идее,

половина максимальной величины типа size_t) функция возвращает код ошибки «нарушено ограничение времени выполнения». Однако поддержка этих функций в стандартной библиотеке С пока еще оставляет желать лучшего.

Константные строки

Следующая программа подготавливает две строки и выводит их на экран:

```
#include <stdio.h>

int main(){
    char *s1 = "Thread";

    char *s2;
    asprintf(&s2, "Floss");

    printf("%s\n", s1);
    printf("%s\n", s2);
}
```

В обоих случаях строка содержит одно слово. Однако компилятор С трактует их совершенно по-разному, и это может застать программиста врасплох.

Вы пробовали выполнить предыдущий пример, в котором запускалась программа поиска строк в исполняемом файле? В данном случае одной из таких строк была бы строка Thread, и переменная s1 указывала бы на область в памяти самой программы. Вот ведь как эффективно – не надо тратить времени, заставляя систему подсчитывать символы, и расходовать память, дублируя информацию, уже присутствующую в исполняемом файле. Полагаю, в 1970-е годы это было важно.

И s1, указывающая на статическую строку, и s2, указывающая на область, динамически выделенную по запросу, с точки зрения чтения, ничем не отличаются, но ни модифицировать, ни освободить s1 нельзя. Ниже показано, что случится, если добавить в программу еще несколько строк:

```
s2[0]='f'; // Первая буква Floss становится маленькой.
s1[0]='t'; // Нарушение защиты памяти.
free(s2);  // Освобождение памяти.
free(s1);  // Нарушение защиты памяти.
```

В зависимости от системы s1 может указывать прямо на строку, являющуюся частью исполняемого файла, или на строку, скопированную в сегмент, допускающий только чтение; на самом деле в C99 §6.4.5(6) и C11 §6.4.5(7) говорится, что способ хранения константных строк не специфицируется, а результат их модификации не определен. Поскольку таким неопределенным поведением может быть – и часто бывает – нарушение защиты памяти, то содержимое s1 надлежит рассматривать как неизменяемое.

Разница между константной и изменяемой строкой тонка и провоцирует ошибки, поэтому зашитые в код строки полезны только в очень специальных контекстах. Не могу представить себе скриптовый язык, в котором это различие было бы существенно.

Однако есть одно простое решение: функция strdup (*string duplicate*), которая определена в стандарте POSIX. Применяется она так:

```
char *s3 = strdup("Thread");
```

Строка Thread по-прежнему зашита в программу, но s3 – копия этой неизменяемой глыбы, поэтому ее уже можно модифицировать, как душа пожелает. Не стесняйтесь пользоваться strdup – и вы сможете обращаться со всеми строками одинаково, не думая о том, какие из них константные, а какие выделены динамически.

Если ваша система не совместима со стандартом POSIX и вас беспокоит отсутствие strdup, то совсем несложно написать ее самостоятельно. Можно, например, воспользоваться все той же asprintf:

```
#ifndef HAVE_STRDUP
char *strdup(char const* in){
    if (!in) return NULL;
    char *out;
    asprintf(&out, "%s", in);
    return out;
}
#endif
```

А откуда берется символ HAVE_STRDUP? Если вы пользуетесь Autotools, то наличие строки

```
AC_CHECK_FUNCS([asprintf strdup])
```

в файле *configure.ac* породило бы в скрипте configure фрагмент, который включает в файл *configure.h* директивы определения или отмены определения HAVE_STRDUP и HAVE_ASPRINTF в зависимости от того, что обнаружено в конкретной системе.

Расширение строк с помощью asprintf

Вот простейший пример добавления текста в существующую строку с помощью функции asprintf:

```
asprintf(&q, "%s и еще фраза%s", q, addme);
```

Такую технику я применяю для построения запросов к базе данных. Я строю запрос из частей, как показано в следующем искусственном примере:

```
int col_number=3, person_number=27;
char *q =strdup("select ");
asprintf(&q, "%scol%i \n", q, col_number);
asprintf(&q, "%sfrom tab \n", q);
asprintf(&q, "%swhere person_id =%i", q, person_number);
```

А в конце получается:

```
select col3
from tab
where person_id = 27
```

Это довольно удобный способ сборки длинной строки, который становится просто необходим по мере усложнения структуры запроса.

Но здесь имеет место утечка памяти, потому что область, на которую первоначально указывала переменная q, не освобождается перед тем, как asprintf запишет в q адрес новой области. Если речь идет об однократном создании строки, то не о чем и беспокоиться – можно насоздавать, не освобождая, несколько миллионов таких запросов, прежде чем случится что-то, достойное внимания.

Если же в программе может быть создано неизвестное число строк неизвестной длины, то следует пользоваться формой, показанной в программе 9.4.

Пример 9.4 ❖ Макрос, который чисто расширяет строки (sasprintf.c)

```
#include <stdio.h>
#include <stdlib.h> //free

// Безопасный макрос, обертывающий asprintf
#define Sasprintf(write_to, ...) { \
   char *tmp_string_for_extend = (write_to); \
   asprintf(&(write_to), __VA_ARGS__); \
   free(tmp_string_for_extend); \
}

// пример использования:
int main(){
   int i=3;
   char *q = NULL;
   Sasprintf(q, "select * from tab");
   Sasprintf(q, "%s where col%i is not null", q, i);
   printf("%s\n", q);
}
```

Макроса Sasprintf в сочетании со strdup может хватить для всего, что приходится делать со строками. Если не считать одной мелочи и необходимости время от времени вызывать free, то можно забыть об управлении памятью.

Мелочь заключается в том, что если вы забудете инициализировать q – значением NULL или посредством strdup, – то при первом обращении макрос Sasprintf освободит случайную область память, на которую указывает неинициализированная переменная q, – вот вам и нарушение защиты.

Например, следующая строка закончится печально – чтобы она заработала, объявление нужно обернуть вызовом strdup:

```
char *q = "select * from"; // ошибка - нужна strdup().
Sasprintf(q, "%s%s where col%i is not null", q, tablename, i);
```

При интенсивном использовании такой способ конкатенации строк теоретически может вызвать замедление работы, поскольку первая часть строки каждый раз перезаписывается. В таком случае используйте C для прототипирования его самого: тогда и только тогда, когда описанная техника оказывается слишком медленной, выделите в своем графике время, чтобы заменить ее использованием snprintf.



Песнь о strtok

Разбиение на лексемы – простейшая и наиболее часто встречающаяся задача разбора текста; цель заключается в том, чтобы разбить строку на части в местах вхождения символов-разделителей. Под это определение подходят разнообразные задачи:

- разбиение на слова, когда в качестве разделителей выступают символы-пустышки: `" \t\n\r"`;
- выделение каталогов из списка путей `"/usr/include:/usr/local/include:."`, в котором каталоги разделены двоеточием;
- разбиение на строчки по символу `"\n"`;
- разбор конфигурационного файла, состоящего из строк вида `ключ=значение`, разделителем в этом случае является знак `"="`;
- список разделенных запятыми значений в файле данных.

Может быть и два уровня разбиения, например сначала конфигурационный файл разбивается на строки по знаку перехода на новую строку, а затем каждая строка разбивается на ключ и значение по знаку =.



Если стоящая перед вами задача сложнее, чем разбиение по одному символу-разделителю, то могут понадобиться регулярные выражения. В разделе «Разбор регулярных выражений» на стр. 321 обсуждается использование совместимых с POSIX регулярных выражений и выделение с их помощью частей строки.

Задача о разбиении на лексемы встречается настолько часто, что в стандартной библиотеке C для нее есть специальная функция, `strtok` (string tokenize), – одна из скромных трудениц, которая хорошо и без лишнего шума делает свою работу.

`strtok` просматривает строку, пока не найдет первый разделитель, после чего записывает на его место `'\0'`. Теперь от строки осталась часть, представляющая первую лексему, и `strtok` возвращает указатель на ее начало. Внутри себя функция хранит информацию об исходной строке, поэтому при следующем вызове она будет искать конец следующей лексемы, заменит очередной разделитель нулем и вернет указатель на начало найденной лексемы.

Начало каждой подстроки – это указатель на позицию внутри уже имеющейся строки, поэтому процесс разбиения сопровождается минимумом операций записи (только символы `\0`) и не требует никакого копирования. Правда, входная строка модифицируется, и поскольку подстроки представляют собой указатели внутрь исходной строки, то исходную строку нельзя освободить, пока не будет закончена работа с подстроками (либо можно с помощью функции `strdup` копировать лексемы в другое место по мере их выделения).

Функция `strtok` хранит указатель на остаток переданной ей строки в единственной внутренней статической переменной-указателе, а это означает, что она может одновременно обрабатывать только одну строку и не предназначена для работы в многопоточной программе. Поэтому функцию `strtok` следует считать нерекомендуемой.

Вместо нее используйте функцию `strtok_r` или `strtok_s` – пригодные для многопоточной работы варианты `strtok`. В стандарте POSIX описана `strtok_r`, а в стан-

дарте C11 – strtok_s. Работать с ними не так удобно, потому что первое обращение отличается от всех последующих.

- При первом обращении к функции в первом аргументе передается разбираемая строка.
- При последующих обращениях в первом аргументе передается NULL.
- Последний аргумент служит для запоминания состояния. При первом обращении инициализировать его не нужно, а при последующих он будет содержать уже разобранную часть строки.

Ниже приведена функция для подсчета строк (точнее, непустых строк; см. предостережение ниже). В скриптовых языках для разбиения на лексему обычно достаточно одной строки кода, но и наша версия, написанная с использованием strtok_r, не намного длиннее. Обратите внимание на конструкцию if ? then : else, необходимую для передачи исходной строки только при первом обращении.

```
#include <string.h> //strtok_r

int count_lines(char *instring){
   int counter = 0;
   char *scratch, *txt, *delimiter = "\n";
   while ((txt = strtok_r(!counter ? instring : NULL, delimiter, &scratch)))
      counter++;
   return counter;
}
```

В разделе о кодировке Unicode приведен полный пример, равно как и в программе из области цетологии в разделе «Подсчет ссылок» на стр. 277.

Функция strtok_s из стандарта C11 работает так же, как strtok_r, но принимает дополнительный аргумент (второй), в котором передается длина входной строки, а после возврата он содержит длину оставшейся части строки. Если входная строка не завершается нулем, то этот дополнительный аргумент будет полезен. Приведенный выше пример можно переписать в таком виде:

```
#include <string.h> //strtok_s

//первое обращение
size_t len = strlen(instring);
txt = strtok_s(instring, &len, delimiter, &scratch);

// последующие обращения:
txt = strtok_s(NULL, &len, delimiter, &scratch);
```



Два и более разделителей подряд рассматриваются как один разделитель, то есть пустые лексемы попросту игнорируются. Например, если разделителем является символ ":" и вы с помощью strtok_r или strtok_s разбираете строку /bin:/usr/bin::/opt/bin, то получите три каталога, потому что :: рассматривается как :. Именно по этой причине показанная выше функция на самом деле подсчитывает непустые строки; ведь два знака перехода на новую строку подряд, как, например, в случае one \n\n three \n four (одна строчка здесь пустая), считаются функцией strtok и ее вариантами как один.

Игнорирование двойных разделителей – часто именно то, что нужно (как в примере со списком путей), но иногда это не так, и в таком случае нужно подумать, как распознавать

двойные разделители. Если разбиваемую строку писали вы сами, то позаботьтесь о специальном маркере для обозначения пустых строк. Написать функцию, которая будет предварительно просматривать строку на предмет обнаружения двойных разделителей, тоже нетрудно (можете также воспользоваться функцией `strsep`, описанной в стандарте BSD/GNU). Если данные поступают от пользователей, то можете повесить суровое предостережение: двойные разделители запрещаются – и предупредить о последствиях, например об игнорировании пустых строк при подсчете.

В примере 9.6 представлена крохотная библиотечка строковых утилит, которые могут вам пригодиться, в том числе несколько уже встречавшихся ранее макросов.

Основных функций две. Первая – `string_from_file` – читает все содержимое файла в одну строку. Это избавляет нас от необходимости читать и обрабатывать небольшие фрагменты файла. Если вы каждодневно работаете с текстовыми файлами гигабайтного размера, то эта функция вам не понадобится, но в случаях, когда размер файла не превышает нескольких мегабайтов, нет смысла возиться с поблочным чтением. Я воспользуюсь этой функцией в нескольких примерах ниже.

Вторая функция `ok_array_new` разбивает строку на лексемы и помещает их в структуру `ok_array`. В примере 9.5 показан заголовок этой библиотеки.

Пример 9.5 ❖ Заголовок библиотеки, содержащей несколько строковых утилит (string_utilities.h)

```
#include <string.h>
#define _GNU_SOURCE //asks stdio.h to include asprintf
#include <stdio.h>

// Безопасный макрос, обертывающий asprintf
#define Sasprintf(write_to, ...) { \                      ❶
    char *tmp_string_for_extend = write_to; \
    asprintf(&(write_to), __VA_ARGS__); \
    free(tmp_string_for_extend); \
}

char *string_from_file(char const *filename);

typedef struct ok_array {
   char **elements;
   char *base_string;
   int length;
} ok_array;                                                ❷

ok_array *ok_array_new(char *instring, char const *delimiters);  ❸

void ok_array_free(ok_array *ok_in);
```

❶ Макрос `Sasprintf`, который мы уже рассматривали, повторен для удобства чтения.

❷ Это массив лексем, который мы получаем, обратившись к функции `ok_array_new` для разбиения строки.

❸ Это обертка над `strtok_r`, которая создает `ok_array`.

В примере 9.6 показано, как с помощью библиотеки GLib прочитать файл в строку и с помощью strtok_r получить из одной строки массив строк. Примеры использования приведены в примерах 9.7, 12.2 и 12.3.

Пример 9.6 ❖ Некоторые полезные строковые утилиты (string_utilities.c)

```
#include <glib.h>
#include <string.h>
#include "string_utilities.h"
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> //abort

char *string_from_file(char const *filename){
   char *out;
   GError *e = NULL;
   GIOChannel *f = g_io_channel_new_file(filename, "r", &e);    ❶
   if (!f) {
      fprintf(stderr, "ошибка при открытии файла '%s'.\n", filename);
      return NULL;
   }
   if (g_io_channel_read_to_end(f, &out, NULL, &e) != G_IO_STATUS_NORMAL){
      fprintf(stderr, "файл '%s' найден, но при его чтении ошибка.\n",
              filename);
      return NULL;
   }
   return out;
}

ok_array *ok_array_new(char *instring, char const *delimiters){    ❷
   ok_array *out= malloc(sizeof(ok_array));
   *out = (ok_array){.base_string=instring};
   char *scratch = NULL;
   char *txt = strtok_r(instring, delimiters, &scratch);
   if (!txt) return NULL;
   while (txt) {
      out->elements = realloc(out->elements, sizeof(char*)*++(out->length));
      out->elements[out->length-1] = txt;
      txt = strtok_r(NULL, delimiters, &scratch);
   }
   return out;
}

/* Освобождает исходную строку, потому что strtok_r ее испортила и
больше она ни на что не годится. */
void ok_array_free(ok_array *ok_in){
   if (ok_in == NULL) return;
   free(ok_in->base_string);
   free(ok_in->elements);
   free(ok_in);
}

#ifdef test_ok_array
int main (){            ❸
```

```
    char *delimiters = " `~!@#$%^&*()_-+={[]}|\\;:\",<>./?\n";
    ok_array *o = ok_array_new(strdup("Hello, reader. This is text."),
                               delimiters);
    assert(o->length==5);
    assert(!strcmp(o->elements[1], "reader"));
    assert(!strcmp(o->elements[4], "text"));
    ok_array_free(o);
    printf("OK.\n");
}
#endif
```

❶ Хотя этот способ годится не всегда, я все же в восторге от возможности просто закачать весь файл в память, переложив заботы с программиста на оборудование. Если файл слишком велик и целиком в файл не помещается, для достижения того же эффекта можно воспользоваться функцией mmap.

❷ Это обертка вокруг strtok_r. Если вы дочитали до этого места, то понимаете, для чего здесь нужен цикл while, а эта функция копирует получаемую на каждой итерации лексему в структуру ok_array.

❸ Если символ test_ok_array не определен, то это библиотека, которую можно использовать в других программах. В противном случае (CFLAGS=-Dtest_ok_array) это исполняемая программа, которая тестирует функцию ok_array_new, предлагая ей разбить конкретную строку по символам, отличающимся от букв и цифр.

Unicode

Во времена, когда все компьютеры были сосредоточены в США, была определена кодировка ASCII (American Standard Code for Information Interchange), сопоставляющая числовые коды всем обычным буквам и символам, имеющимся на клавиатуре US QWERTY. Этот набор символов я буду называть наивным английским. Тип C char занимает 8 бит (двоичных цифр), или 1 байт, то есть может представлять 256 различных значений. В кодировке ASCII определено 128 символов, так что все они представимы типом char, и один бит еще остается в запасе. Этот восьмой бит во всех символах ASCII равен 0, что оказалось весьма кстати.

В кодировке Unicode применены те же основные принципы, только каждому *глифу* (визуальному образу, понятному людям) сопоставляется шестнадцатеричный код, как правило, из диапазона от 0000 до FFFF[1]. Принято обозначать такие

---

[1] Диапазон от 0000 до FFFF называется основным многоязычным уровнем (ОМУ, или BMP) и включает большинство, но не все символы, встречающиеся в современных языках. Другие кодовые позиции (теоретически от 10000 до 10FFFF) находятся на дополнительных уровнях. Здесь размещены математические символы (например, символ множества вещественных чисел ℝ) и унифицированный набор идеограмм китайского, японского и корейского языков (CJK). Если вы один из десяти миллионов китайцев народности мяо или один из сотен или тысяч говорящих на распространенных в Индии языках сора или чакма, то вам сюда. Да, подавляющее большинство тестов можно записать символами из уровня BMP, но заверяю вас – предположение о том, что в любом тексте встречаются только символы с кодами меньше FFFF, заведомо неверно.

кодовые позиции в виде U+0000. Работа по созданию кодировки Unicode оказалась куда более трудной, потому что пришлось каталогизировать все обычные буквы западных алфавитов, десятки тысяч китайских и японских иероглифов, все обязательные глифы угаритского и дезеретского алфавита и т. д. и т. п. – для всего земного шара на протяжении всей истории человечества. Следующий вопрос – как все это закодировать, и вот тут начинаются расхождения. Главный вопрос – сколько байтов должна занимать единица, подвергающаяся анализу. В UTF-32 (UTF расшифровывается как UCS Transformation Format – формат преобразования UCS, а UCS – как Universal Character Set – универсальный набор символов) основной единицей считаются 32 бита, или 4 байта, а это означает, что каждый символ можно закодировать одной единицей, но ценой гигантского перерасхода памяти на дополнение до 4 байтов, ведь в наивном английском наборе каждый символ представлен всего 7 битами. В UTF-16 основная единица составляет 2 байта, и этого хватает для представления большинства символов одной единицей, но некоторые приходится кодировать двумя. В UTF-8 в качестве единицы используется 1 байт, и, следовательно, еще больше кодовых позиций приходится кодировать несколькими единицами.

Я предпочитаю рассматривать кодировки UTF как вариант тривиального шифрования. Для каждой кодовой позиции существует единственная последовательность байтов в UTF-8, единственная последовательность байтов в UTF-16 и единственная последовательность байтов в UTF-32, и все они имеют между собой мало общего. Если отвлечься от обсуждаемого ниже исключения, то нет никаких причин ожидать, что кодовая позиция и любое из его зашифрованных представлений будут численно одинаковы или вообще связаны хоть каким-то очевидным образом, но я знаю, что правильно написанный программный декодер сможет легко и однозначно выполнить переход от любой кодировки UTF к правильной кодовой позиции Unicode.

А что используется на разных компьютерах? В веб есть непререкаемый лидер: в настоящее время свыше 80% сайтов пользуются кодировкой UTF-8[1]. Mac и Linux также по умолчанию всегда применяют UTF-8, так что есть все шансы, что никак не помеченный текстовый файл на машине под управлением Mac или Linux представлен в кодировке UTF-8.

Примерно 10% сайтов в мире еще не перешли на Unicode, а используют довольно архаичную систему ISO/IEC 8859 (в которой имеются кодовые страницы, например Latin-1). А в Windows – свободомыслящей операционной системе, которая посылает POSIX на три буквы, – применяется UTF-16.

Отображение текстов в кодировке Unicode – дело операционной системы, и дело непростое. Например, при печати наивного английского набора символов каждый символ занимает одну позицию в строке текста, но, к примеру, буква иврита בּ = b записывается в виде комбинации двух кодовых позиций ב (U+05D1) и (U+05BC). Для построения составного символа к согласным добавляются глас-

[1] http://w3techs.com/technologies/overview/character_encoding/all.

ные: בָּ = ba(U+05D1 + U+05BC + U+05B8). А сколько байтов потребуется для кодирования трех этих кодовых позиций в UTF-8 (в данном случае шесть) – отдельный, совершенно независимый вопрос. Говоря же о длине строки, мы можем иметь в виду число кодовых позиций, ширину на экране или количество байтов, необходимых для представления строки.

И каковы же обязанности автора, который желает сделать свою программу понятной всем говорящим на столь разных языках? Вы должны:

- определить, какую кодировку использует целевая система, чтобы случайно не начать читать входные данные в другой кодировке, и выводить результаты в виде, который система сможет декодировать;
- где-то сохранять текст в неизменном виде;
- понимать, что один символ необязательно представляется фиксированным числом байтов, поэтому, произвольно сместившись от начала строки, можно оказаться в середине кодовой позиции (если дана Unicode-строка `us`, то к такой неприятности может привести, например, операция `us++`);
- иметь под рукой средства для различных манипуляций с текстом, `toupper` и `tolower` работают только для наивного английского, так что нужны альтернативы.

Чтобы удовлетворить эти требования, вы должны будете выбрать правильную внутреннюю кодировку, чтобы не исказить текст, и располагать хорошей библиотекой, помогающей в деле декодирования.

Кодировка для программ на С

Выбрать внутреннюю кодировку совсем просто. UTF-8 была спроектирована специально для нас, программистов на С.

- Единица в UTF-8 составляет 8 бит: как раз длина `char`[1]. Допустимо представлять строку в кодировке UTF-8 как `char *` – точно так же, как наивный английский текст.
- Первые 128 кодовых позиций Unicode в точности совпадают с кодировкой ASCII. Например, букве *A* соответствуют код 41 (шестнадцатеричный) в ASCII и кодовая позиция U+0041 в Unicode. Поэтому если Unicode-текст состоит только из наивных английских символов, то при работе с ним вполне можно использовать как обычные средства, ориентированные на ASCII, так и средства для работы с UTF-8. Это возможно благодаря нехитрому трюку: если восьмой бит `char` равен 0, то `char` представляет символ ASCII, а если 1, то это один из байтов многобайтного символа. Таким образом, никакая часть символа Unicode, которому нет соответствия ASCII, в кодировке UTF-8 не будет совпадать ни с одним символом ASCII.
- U+0000 – допустимая кодовая позиция, которую мы, пишущие на С, обозначаем как `'\0'`. Поскольку \0 одновременно является и нулем в кодировке

[1] Быть может, когда-то и существовали ориентированные на ASCII машины, в которых длина char составляла 7 бит, но в стандартах C99 и C11 §5.2.4.2.1(1) четко определено, что значение CHAR_BIT должно быть не меньше 8; см. также §6.2.6.1(4), где байт определен как CHAR_BIT битов.

ASCII, то это правило – частный случай предыдущего. Это важно, потому что строка UTF-8 с одним символом \0 в конце – это именно то, что считается в C правильной строкой char *. Вспомните, что в кодировках UTF-16 и UTF-32 символ занимает несколько байтов, поэтому символы наивного английского набора дополняются до минимальной единицы; это означает, что первые 8 бит с большой вероятностью будут нулевыми, а следовательно, сохранение текста в кодировке UTF-16 или UTF-32 в переменной типа char * даст строку, испещренную нулевыми байтами.

Итак, о пишущих на C позаботились: закодированный в UTF-8 текст можно хранить и копировать, пользуясь давно привычным нам типом строки char *. Однако теперь, когда один символ может занимать несколько байтов, нужно следить за тем, чтобы не изменить порядка байтов и ни в коем случае не расщепить многобайтный символ. Если ничего такого вы не делаете, то можете спокойно работать со строкой, как будто она состоит из наивных английских символов. Ниже приведен неполный перечень функций из стандартной библиотеки, безопасных относительно UTF-8:

- strdup и strndup;
- strcat и strncat;
- strcpy и strncpy;
- определенные в POSIX basename и dirname;
- strcmp и strncmp, но только для сравнения строк на равенство (нуль или не нуль). Для осмысленной сортировки понадобятся функции, учитывающие порядок символов (см. следующий раздел);
- strstr;
- printf и прочие функции из этого семейства, включая sprintf, причем спецификатор%s по-прежнему применяется для подстановки строки;
- strtok_r, strtok_s и strsep при условии, что разбиение производится по ASCII-символу, например " \t\n\r:|;,";
- strlen и strnlen, но помните, что возвращается число байтов, а не число кодовых позиций и не ширина, занимаемая на экране. Для этих целей необходима новая библиотечная функция, обсуждаемая в следующем разделе.

Все эти функции работают на уровне байтов, но для большинства содержательных операций с текстом требуется его декодировать. И это подводит нас к библиотекам.

Библиотеки для работы с Unicode

Первое, что мы должны сделать, – преобразовать данные, пришедшие из внешнего мира, в UTF-8, чтобы можно было с ними работать. А значит, нужны функции-привратники, которые кодируют входящие символы в UTF-8 и перекодируют исходящие строки из UTF-8 в кодировку, ожидаемую потребителем. Они дают нам возможность внутри программы работать с одной разумной кодировкой.

Именно так работает библиотека Libxml (см. раздел «libxml and cURL» ниже): в правильном XML-документе используемая кодировка указана в заголовке (но

в библиотеке есть набор правил для угадывания кодировки, если ее объявление отсутствует), поэтому Libxml знает, как производить перекодирование. Libxml разбирает документ и преобразует его во внутренний формат, допускающий запросы и редактирование. Отвлекаясь от возможных ошибок, можно быть уверенным, что внутренний формат представлен в кодировке UTF-8, потому что Libxml отказывается работать с любыми другими.

Если вы собираетесь самостоятельно производить перекодирование на входе, то в вашем распоряжении имеется описанная в POSIX функция `iconv`. С учетом того, сколько кодировок эта функция должна обрабатывать, она, надо полагать, невообразимо сложна. На случай, если в вашей системе ее нет, GNU предлагает переносимую библиотеку libiconv.

 В стандарте POSIX описана также командная утилита `iconv`, обертывающая эту функцию.

В библиотеке GLib есть несколько оберток вокруг `iconv`, нас больше всего интересуют `g_locale_to_utf8` и `g_locale_from_utf8`. А в руководстве по GLib вы найдете длинный раздел, посвященный средствам манипуляции Unicode-текстами. Вы увидите, что есть два класса таких средств: для UTF-8 и для UTF-32 (для последней в GLib предусмотрен тип `gunichar`).

Напомним, что 8 байтов даже близко не хватает для представления всех символов одной единицей, поэтому одному символу может соответствовать от одной до шести единиц. Из-за этого UTF-8 называется *многобайтной кодировкой*, и это же влечет за собой проблемы с вычислением истинной длины строки (где под *длиной* понимается количество символов или ширина на экране), получением следующего полного символа, получением подстроки или сравнением с целью сортировки (или *упорядочением* – collating).

В UTF-32 применяется дополнение, так чтобы все символы можно было представить одним и тем же числом единиц, отсюда и название *широкий символ*. Нередко встречается термин «преобразование из многобайтной кодировки в широкую» – теперь вы знаете, что это такое.

Имея один символ в кодировке UTF-32 (тип GLib `gunichar`), вы можете безо всяких проблем делать с ним все, что принято делать с символами: получить тип (буква, цифра и т. д.), преобразовать регистр и т. д.

Если вы читали стандарт C, то несомненно обратили внимание, что там описан тип широкого символа и все сопутствующие ему функции. Тип `wchar_t` появился еще в стандарте C89, то есть до публикации первой версии стандарта Unicode. Не думаю, что он до сих пор полезен. Ширина `wchar_t` в стандарте не оговорена, поэтому может быть равна и 32 бита, и 16 бит (и вообще какая угодно). Компиляторы в Windows считают, что она равна 16 бит, чтобы не противоречить предпочтению, которое Microsoft отдает кодировке UTF-16, но UTF-16 – это все же многобайтная кодировка, поэтому необходим еще один тип, гарантирующий кодирование с постоянной шириной символа. В C11 проблема исправлена за счет добавления типов `char16_t` и `char32_t`, но пока что программ, в которых они используются, совсем немного.

Пример кода

В примере 9.7 приведена программа, которая читает файл и разбивает его на «слова»; под этим я понимаю применение strtok_r для разбиения на лексемы по пробелам и символам новой строки – широко распространенное определение. Для каждого слова я с помощью GLib преобразую первый символ из многобайтной кодировки UTF-8 в широкую кодировку UTF-32, а затем печатаю информацию о том, что он собой представляет: букву, цифру или широкий символ CJK (то есть китайский, японский или корейский иероглиф, для печати которых обычно отводится больше места).

Функция string_from_file читает все содержимое файла в строку, а затем local_string_to_utf8 преобразует его из локальной кодировки компьютера в UTF-8. В моем использовании strtok_r есть только примечательная особенность – тот факт, что ничего примечательного в нем нет. Если разбиение производится по пробелам и знакам новой строки, то гарантированно не произойдет расщепления многобайтного символа на две части.

Я вывожу результат в формате HTML, потому что в этом случае могу указать кодировку UTF-8 и не думать о кодировании на выходе. Если ваша операционная система работает в UTF-16, откройте результирующий файл в браузере.

Поскольку в этой программе используются библиотеки GLib и string_utilities, мой makefile выглядит следующим образом:

```
CFLAGS==`pkg-config --cflags glib-2.0` -g -Wall -O3
LDADD=`pkg-config --libs glib-2.0`
CC=c99
objects=string_utilities.o

unicode: $(objects)
```

В примере 10.21 приведена еще одна программа, работающая с символами Unicode, она перечисляет все символы во всех находящихся в некотором каталоге файлах в кодировке UTF-8.

Пример 9.7 ❖ **Прочитать текстовый файл и вывести полезную информацию о встретившихся символах (unicode.c)**

```
#include <glib.h>
#include <locale.h> //setlocale
#include "string_utilities.h"
#include "stopif.h"

// Освобождает instring, больше ни для чего полезного ее
// использовать нельзя.
char *localstring_to_utf8(char *instring){                          ❶
   GError *e=NULL;
   setlocale(LC_ALL, ""); //получить локаль ОС
   char *out = g_locale_to_utf8(instring, -1, NULL, NULL, &e);
   free(instring); // оригинал больше не нужен
   Stopif(!g_utf8_validate(out, -1, NULL), free(out); return NULL,
      "Ошибка: не удалось преобразовать ваш файл в строку UTF-8.");
```

```
   return out;
}

int main(int argc, char **argv){
   Stopif(argc==1, return 1, "Укажите в аргументе имя файла. "
      "Я выведу полезную информацию о нем в uout.html.");
   char *ucs = localstring_to_utf8(string_from_file(argv[1]));
   Stopif(!ucs, return 1, "Exiting.");
   FILE *out = fopen("uout.html", "w");
      Stopif(!out, return 1, "Не удалось открыть uout.html для записи.");
   fprintf(out, "<head><meta http-equiv=\"Content-Type\" "
      "content=\"text/html; charset=UTF-8\" />\n");
   fprintf(out, "В этом документе%li символов.<br>", g_utf8_strlen(ucs, -1));   ❷
   fprintf(out, "Для кодироввания в Unicode потребовалось%zu байтов.<br>",
      strlen(ucs));
   fprintf(out, "Вот он, по одному ограниченному пробелами элементу "
         "в строке (с комментарием о первом символе):<br>");
   ok_array *spaced = ok_array_new(ucs, " \n");                                 ❸
   for (int i=0; i< spaced->length; i++, (spaced->elements)++){
      fprintf(out, "%s", *spaced->elements);
      gunichar c = g_utf8_get_char(*spaced->elements);                          ❹
      if (g_unichar_isalpha(c)) fprintf(out, " (a letter)");
      if (g_unichar_isdigit(c)) fprintf(out, " (a digit)");
      if (g_unichar_iswide(c)) fprintf(out, " (wide, CJK)");
      fprintf(out, "<br>");
   }
   fclose(out);
   printf("Информация выведена в uout.html. Откройте файл в браузере.\n");
}
```

❶ Это привратник на входе, он преобразует символы из локальной кодировки компьютера в UTF-8. Привратника на выходе нет, потому что я пишу в HTML-файл, а браузеры знают, как поступать с UTF-8. Привратник на выходе был бы очень похож на эту функцию, только вызывал бы `g_locale_from_utf8`.

❷ `strlen` принадлежит к функциям, которые считают, что в каждом символе ровно один байт, поэтому ее необходимо заменить.

❸ Рассмотренная выше функция `ok_array_new` разбивает текст по пробелам и знакам перехода на новую строку.

❹ Здесь мы видим несколько операций над одним символом, которые работают только после преобразования из многобайтной кодировки UTF-8 в широкую кодировку с постоянной шириной символа.

Gettext

Наверное, ваша программа выводит множество сообщений пользователям, например сообщения об ошибках и приглашения. По-настоящему дружественная к пользователю программа переводит эти тексты на разные языки – чем больше, тем лучше. Инструмент GNU Gettext предлагает систему для организации переводов. Руководство по Gettext написано довольно хорошо, поэтому за деталями отсылаю к нему, а здесь скажу лишь пару слов, чтобы вы могли составить себе представление о процедуре в целом.

- Замените все сообщения вида `"Human message"` в программе на `_("Human message")`. Знак подчеркивания – это макрос, который в конечном итоге расширяется в вызов функ-

ции, выбирающей строку, соответствующую локали компьютера, на котором исполняется программа.

- Выполните программу `xgettext`, которая составит индекс строк, нуждающихся в переводе, в виде переносимого объектного шаблона – *pot*-файла.
- Отправьте этот *pot*-файл коллегам, говорящим на других языках, и попросите их прислать *po*-файлы с переводом строк на свой родной язык.
- Включите в свой файл *configure.ac* макрос `AM_GNU_GETTEXT` (наряду с другими необязательными макросами, определяющими, где искать *po*-файлы и другие детали такого рода).

Глава 10

Улучшенная структура

Двадцать девять разных признаков,
и только семь из них тебе нравятся.
— The Strokes «You Only Live Once»

Это глава о функциях, которые принимают на входе структуры, и о том, как улучшить интерфейс с библиотекой.

Мы начнем с трех синтаксических новаций, включенных в стандарт ISO C99: составные литералы, макросы с переменным числом аргументов и позиционные инициализаторы. По существу, эта глава представляет собой развернутое обсуждение того, что можно сделать с помощью различных комбинаций этих трех элементов.

Составные литералы, взятые сами по себе, упрощают передачу списка функции. Макросы с переменным числом аргументов позволяют скрыть синтаксис составных литералов от пользователя, который видит только функцию, принимающую список произвольной длины: и `f(1, 2)`, и `f(1, 2, 3, 4)` допустимы.

С помощью похожего трюка можно было бы реализовать ключевое слово `foreach`, встречающееся во многих языках, или векторизовать функцию с одним аргументом, так чтобы она могла применяться к нескольким аргументам.

Позиционные инициализаторы существенно упрощают работу со структурами – настолько, что я почти перестал пользоваться старым способом. Вместо непонятной и чреватой ошибками конструкции `person_struct p = {"Joe", 22, 75, 20}` мы можем писать самодокументированные объявления вида `person_struct p = {.name="Joe", .age=22, .weight_kg=75, .education_years=20}`.

Ну а коль скоро инициализация структуры перестала быть занозой, то и возврат структуры из функции оказывается столь же простым, и это позволяет сделать интерфейс функции прозрачнее.

Передача структуры функции также стала проще. Обернув все еще одним макросом с переменным числом аргументов, мы теперь можем писать функции, принимающие переменное число именованных аргументов, и даже присваивать значения по умолчанию тем аргументам, которые пользователь не задал. В примере программы кредитного калькулятора имеется функция, которую можно вызывать и так: `amortization(.amount=200000, .rate=4.5, .years=30)`, и так: `amortization(.rate=4.5, .amount=200000)`. Поскольку во втором случае срок кредита не указан, функция по умолчанию подставляет 30 лет.

Далее в этой главе будут приведены примеры ситуаций, в которых входные и выходные структуры могут облегчить жизнь, в том числе при работе с функциями, интерфейс которых основан на указателях на void, а также когда унаследованный код имеет столь чудовищный интерфейс, что его хорошо бы обернуть чем-то более удобным.

Составные литералы

Передать функции литеральное значение ничего не составляет: если имеется объявление `double a_value`, то C прекрасно понимает, что такое `f(a_value)`.

Но если вы хотите передать список элементов – составное литеральное значение вида `{20.38, a_value, 9.8}`, – то возникает синтаксическое неудобство: требуется поставить перед составным литералом оператор приведения типа, иначе анализатор не поймет, что имеется в виду. Теперь список выглядит как `(double[]) {20.38, a_value, 9.8}`, а обращение принимает вид:

```
f((double[]) {20.38, a_value, 9.8});
```

Память для составных литералов выделяется автоматически, то есть ни `malloc`, ни `free` не нужны. В конце области видимости, в которой появился составной литерал, он попросту исчезает.

Пример 10.1 начинается довольно типичной функцией `sum`, которая принимает массив элементов типа `double` и вычисляет их сумму, пока не встретится первый элемент NaN (нечисло, см. раздел «Пометка недопустимых числовых значений с помощью маркера NaN» выше). Если во входном массиве нет элементов NaN, то результат будет катастрофическим; позже мы добавим меры предосторожности. Из `main` эта функция вызывается двумя способами: традиционным – с использованием временной переменной – и с помощью составного литерала.

Пример 10.1 ❖ Мы можем обойтись без временной переменной, воспользовавшись составным литералом (sum_to_nan.c)

```
#include <math.h> //NAN
#include <stdio.h>

double sum(double in[]){                                      ❶
  double out=0;
  for (int i=0; !isnan(in[i]); i++) out += in[i];
  return out;
}

int main(){
  double list[] = {1.1, 2.2, 3.3, NAN};                       ❷
  printf("sum:%g\n", sum(list));

  printf("sum:%g\n", sum((double[]){1.1, 2.2, 3.3, NAN}));    ❸
}
```

❶ Эта ничем не примечательная функция складывает элементы входного массива, пока не дойдет до первого маркера NaN.

❷ Это типичный случай использования функции, которая принимает на входе массив. В первой строчке мы объявляем временную переменную, содержащую список, а во второй передаем эту переменную функции.
❸ Здесь мы делаем то же самое без заведения промежуточной переменной – для инициализации массива используется составной литерал, который передается функции непосредственно.

Это простейший пример использования составных литералов; далее в этой главе мы увидим, как они применяются и для других целей. А пока проверьте, нет ли в ваших программах мест, которые можно было бы упростить за счет инициализации однократно используемых списков составными литералами.



В этом примере создается массив в автоматической памяти, а не указатель на массив, поэтому параметр функции следует описывать как имеющий тип (double[]), а не (double*).

Инициализация с помощью составных литералов

Остановимся на одном тонком различии, которое, возможно, поможет вам лучше понять принцип работы составных литералов.

Надо полагать, вы давно привыкли объявлять массивы следующим образом:

```
double list[] = {1.1, 2.2, 3.3, NAN};
```

Здесь выделяется память для именованного массива list. Результат вычисления sizeof(list) равен 4 * sizeof(double). Иными словами, list – это самый *настоящий* массив (в смысле, описанном в разделе «Автоматическая, статическая и динамическая память» выше).

Можно было бы воспользоваться для объявления составным литералом, о наличии которого свидетельствует приведение к типу (double[]):

```
double *list = (double[]){1.1, 2.2, 3.3, NAN};
```

При этом система сначала генерирует анонимный список, вставляет его в кадр стека, принадлежащий функции, а затем объявляет указатель list на этот анонимный список. Таким образом, list оказывается *псевдонимом*, а sizeof(list) равно sizeof(double*). Это наглядно продемонстрировано в примере 8.2.

Макросы с переменным числом аргументов

Вообще говоря, я считаю, что механизм функций с переменным числом аргументов в языке C дефектный (подробнее об этом см. в разделе «Гибкая передача аргументов функциям» ниже). Однако с макросами такой проблемы не возникает. Ключевое слово __VA_ARGS__ расширяется в переданный набор аргументов.

Пример 10.2 – это повторение примера 2.5 – специализированный интерфейс к функции printf, которая печатает сообщение, если утверждение ложно.

Пример 10.2 ❖ Тот же макрос для обработки ошибок, что в примере 2.5 (stopif.h)

```
#include <stdio.h>
#include <stdlib.h> //abort

/** Присвоить этой переменной значение \c 's', если нужно, чтобы программа
    останавливалась после ошибки.
    В противном случае программа будет возвращать код ошибки.*/
char error_mode;

/** Куда писать сообщения об ошибках? Если \c NULL, писать в \c stderr. */
FILE *error_log;

#define Stopif(assertion, error_action, ...) {\
       if (assertion){    \
          fprintf(error_log ? error_log : stderr, __VA_ARGS__);  \
          fprintf(error_log ? error_log : stderr, "\n"); \
          if (error_mode=='s') abort(); \
          else                 {error_action;} \
       } }

// пример использования:
Stopif(x<0 || x>1, return -1, "значение x равно%g, "
"а должно быть числом от нуля до единицы.", x);
```

Все, что пользователь напишет на месте многоточия (...), будет подставлено вместо маркера `__VA_ARGS__`.

Чтобы продемонстрировать потенциальные возможности макросов с переменным числом аргументов, мы в примере 10.3 реализовали синтаксис цикла `for`. Все находящееся после второго аргумента – сколько бы запятых там ни было – считается аргументом `...` и подставляется вместо маркера `__VA_ARGS__`.

Пример 10.3 ❖ Многоточие ... в макросе скрывает за собой все тело цикла `for` (varad.c)

```
#include <stdio.h>

#define forloop(i, loopmax, ...) for(int i=0; i< loopmax; i++) \
                                  {__VA_ARGS__}

int main(){
   int sum=0;
   forloop(i, 10,
           sum += i;
           printf("sum to%i:%i\n", i, sum);
   )
}
```

Я бы не стал пользоваться кодом из примера 10.3 в реальной программе, но лишь слегка различающиеся фрагменты встречаются достаточно часто, и иногда для устранения избыточности стоит воспользоваться макросами с переменным числом аргументов.

Безопасное завершение списков

Составные литералы и макросы с переменным числом аргументов – сладкая парочка, позволяющая применять макросы для построения списков и структур. К структурам мы перейдем чуть позже, а пока займемся списками.

Пару страниц назад мы видели функцию, которая принимает список и складывает его элементы, пока не встретится первый маркер NaN. Для использования этой функции знать длину входного массива необязательно, но должна быть гарантия, что в конце его находится элемент NaN; если это не так, ждите нарушения защиты памяти. Гарантировать наличие завершающего маркера NaN можно путем вызова `sum` через макрос с переменным числом аргументов, как в примере 10.4.

Пример 10.4 ❖ Использование макроса с переменным числом аргументов для порождения составного литерала (safe_sum.c)

```
#include <math.h> //NAN
#include <stdio.h>

double sum_array(double in[]){                              ❶
   double out=0;
   for (int i=0; !isnan(in[i]); i++) out += in[i];
   return out;
}

#define sum(...) sum_array((double[]){__VA_ARGS__, NAN})    ❷

int main(){
   double two_and_two = sum(2, 2);                          ❸
   printf("2+2 =%g\n", two_and_two);
   printf("(2+2)*3 =%g\n", sum(two_and_two, two_and_two, two_and_two));
   printf("sum(asst) =%g\n", sum(3.1415, two_and_two, 3, 8, 98.4));
}
```

❶ Функция называется иначе, но по-прежнему занимается суммированием элементов массива.

❷ Именно в этой строке происходит самое интересное: макрос с переменным числом аргументов копирует свои аргументы в составной литерал. Таким образом, макрос принимает произвольное количество значений типа `double`, а передает их в виде одного списка, в конце которого гарантированно находится NaN.

❸ Теперь `main` может передавать макросу `sum` списки чисел произвольной длины, а о добавлении завершающего маркера NaN пусть заботится сам макрос.

Думаю, вы согласитесь, что это изящная функция. Она принимает столько входных параметров, сколько у вас есть, не заставляя предварительно упаковывать их в массив. Достигается это за счет того, что макрос пользуется составным литералом для инициализации массива.

На самом деле макрос работает только с отдельными числами, массивов он не понимает. Если у вас уже есть массив и вы уверены, что он завершается маркером `NaN`, то вызывайте функцию `sum_array` напрямую.

Несколько списков

А что, если требуется передать *два* списка произвольной длины? Пусть, например, программа должна извещать об ошибках двумя способами: печатать понятное человеку сообщение и записывать машиночитаемый код ошибки в журнал (я буду использовать в этом качестве stderr). Хорошо бы иметь одну функцию, принимающую аргументы для обеих функций вывода в духе printf, но как компилятор поймет, где заканчивается один набор аргументов и начинается следующий?

Можно сгруппировать аргументы, как мы это всегда делаем: с помощью скобок. При обращении к макросу my_macro вида my_macro(f(a, b), c) первым аргументом будет все выражение f(a, b) – запятая внутри скобок не считается разделителем аргументов макроса, потому что в таком случае была бы нарушена семантика скобок и получилась бы чепуха [C99 и C11 §6.10.3(11)].

А раз так, то можно следующим образом написать код, печатающий сразу два сообщения об ошибке:

```
#define fileprintf(...) fprintf(stderr, __VA_ARGS__)
#define doubleprintf(human, machine) \
do {printf human; fileprintf machine;} while(0)

// пример использования:
if (x < 0)
   doubleprintf(("x отрицательно (равно:%g)\n", x), ("NEGVAL: x=%g\n", x));
```

Результатом расширения этого макроса является такой код:

```
do {printf ("x is отрицательно (равно:%g)\n", x);\
    fileprintf ("NEGVAL: x=%g\n", x);}
while(0);
```

Я добавил макрос fileprintf для единообразия. Не будь его, аргументы printf для вывода сообщения человеку нужно было бы заключать в скобки, а аргументы printf для вывода в журнал – оставить вне скобок:

```
#define doubleprintf(human, ...) do {printf human;\
                                    fprintf (stderr, __VA_ARGS__);} while(0)

// и затем так:
if (x < 0)
   doubleprintf(("x меньше нуля (равно:%g)\n", x), "NEGVAL: x=%g\n", x);
```

Это вполне корректный синтаксис, но мне он не нравится, потому что функционально однотипные вещи должны и выглядеть одинаково.

А если пользователь вообще забудет про скобки? Тогда код не откомпилируется: почти любой символ после printf, кроме открывающей круглой скобки, приведет к загадочному сообщению об ошибке. Итак, с одной стороны, непонятное сообщение, зато с другой – невозможность случайно забыть скобки и тем самым поставить заказчику неправильный код.

В примере 10.5 показано еще одно применение макросов: если даны два списка R и C, то элемент (i, j) будет содержать произведение $R_i C_j$. В основе этого примера лежит макрос matrix_cross с достаточно удобным интерфейсом.

Пример 10.5 ❖ Передача функции двух списков переменной длины (times_table.c)

```
#include <math.h> //NAN
#include <stdio.h>

#define make_a_list(...) (double[]){__VA_ARGS__, NAN}

#define matrix_cross(list1, list2) matrix_cross_base(make_a_list list1, \
                    make_a_list list2)

void matrix_cross_base(double *list1, double *list2){
   int count1 = 0, count2 = 0;
   while (!isnan(list1[count1])) count1++;
   while (!isnan(list2[count2])) count2++;
   if (!count1 || !count2) {printf("отсутствуют данные."); return;}

   for (int i=0; i<count1; i++){
      for (int j=0; j<count2; j++)
         printf("%g\t", list1[i]*list2[j]);
      printf("\n");
   }
   printf("\n\n");
}

int main(){
   matrix_cross((1, 2, 4, 8), (5, 11.11, 15));
   matrix_cross((17, 19, 23), (1, 2, 3, 5, 7, 11, 13));
   matrix_cross((1, 2, 3, 5, 7, 11, 13), (1)); // вектор столбцов
}
```

Foreach

Как мы уже видели, составной литерал можно использовать всюду, где допустим массив или структура. Вот, например, как с помощью составного литерала объявляется массив строк:

```
char **strings = (char*[]){"Yarn", "twine"};
```

Теперь обернем это циклом for. В первом компоненте цикла объявляется массив строк, так что мы можем воспользоваться показанным выше предложением. Затем мы выполняем итерации, пока не достигнем завершающего маркера NULL. Чтобы было понятнее, я ввел typedef для типа строки:

```
#include <stdio.h>

typedef char* string;

int main(){
   string str = "thread";
   for (string *list = (string[]){"yarn", str, "rope", NULL}; *list; list++)

      printf("%s\n", *list);
}
```

Это еще чересчур многословно, поэтому уберем синтаксический шум внутрь макроса. Вот теперь main оказывается настолько чистой, насколько возможно:

```
#include <stdio.h>

// На этот раз я обошелся без typedef.
#define Foreach_string(iterator, ...)\
for (char **iterator = (char*[]){__VA_ARGS__, NULL}; *iterator;

int main(){
char *str = "thread";
   Foreach_string(i, "yarn", str, "rope"){
      printf("%s\n", *i);
   }
}
```

Векторизация функции

Функция free принимает ровно один аргумент, поэтому часто код очистки в конце функции включает длинные цепочки вида:

```
free(ptr1);
free(ptr2);
free(ptr3);
free(ptr4);
```

Ну раздражает же! Ни один уважающий себя программист на LISP не потерпел бы такой избыточности, а написал бы векторизованную функцию free, которую можно было бы вызывать так:

```
free_all(ptr1, ptr2, ptr3, ptr4);
```

Если вы дочитали до этого места, то следующая фраза не вызовет недоумения: мы можем написать макрос с переменным числом аргументов, который генерирует массив (с завершающим маркером) с помощью составного литерала, а затем выполняет цикл for, в котором функция применяется к каждому элементу массива. В примере 10.6 показана реализация этой идеи.

Пример 10.6 ❖ Техника векторизации произвольной функции, принимающей указатель любого типа (vectorize.c)

```
#include <stdio.h>
#include <stdlib.h> //malloc, free

#define Fn_apply(type, fn, ...) {  \           ❶
   void *stopper_for_apply = (int[]){0}; \    ❷
   type **list_for_apply = (type*[]){__VA_ARGS__, stopper_for_apply};  \
   for (int i=0; list_for_apply[i] != stopper_for_apply; i++)  \
      fn(list_for_apply[i]);  \
}

#define Free_all(...) Fn_apply(void, free, __VA_ARGS__);

int main(){
```

```
    double *x= malloc(10);
    double *y= malloc(100);
    double *z= malloc(1000);
    Free_all(x, y, z);
}
```

❶ Для пущей безопасности макрос принимает имя типа. Я поставил его раньше имени функции, потому что порядок «тип, затем имя» принят и при объявлении функций.

❷ Необходим завершающий маркер, который заведомо не совпадает ни с одним из используемых в программе указателей, в том числе NULL, поэтому мы воспользуемся составным литералом, чтобы инициализировать массив из одного целого числа, и укажем на этот массив. Обратите внимание, что в условии окончания цикла проверяется сам указатель, а не значение, на которое он указывает.

Подготовив все необходимое, мы можем обернуть этим векторизующим макросом любую функцию, принимающую один указатель. При работе с библиотекой GSL можно было бы определить:

```
#define Gsl_vector_free_all(...) \
    Fn_apply(gsl_vector, gsl_vector_free, __VA_ARGS__);
#define Gsl_matrix_free_all(...) \
    Fn_apply(gsl_matrix, gsl_matrix_free, __VA_ARGS__);
```

Мы не отказываемся от проверки типов на этапе компиляции (если только тип указателя не void), а это гарантирует, что на вход макросу подается список указателей одного и того же типа. Если необходимы разнородные элементы, то понадобится еще один прием – позиционные инициализаторы.

Позиционные инициализаторы

Эту идею я проиллюстрирую на примере. Ниже приведена короткая программа, которая выводит на экран решетку 3×3 и в одной ячейке печатает звездочку. Местоположение звездочки – правая верхняя ячейка, левая средняя и т. д. – задается в структуре direction_s.

В примере 10.7 наибольший интерес представляет функция main, в которой объявлены три такие структуры с помощью позиционных инициализаторов, – мы указываем в инициализаторе имена элементов структуры.

Пример 10.7 ❖ Задание структуры с помощью позиционных инициализаторов (boxes.c)

```
#include <stdio.h>

typedef struct {
    char *name;
    int left, right, up, down;
} direction_s;

void this_row(direction_s d); // эти функции находятся ниже
```

```
void draw_box(direction_s d);

int main(){
   direction_s D = {.name="left", .left=1};                ❶
   draw_box(D);

   D = (direction_s) {"upper right", .up=1, .right=1};    ❷
   draw_box(D);

   draw_box((direction_s){});                               ❸
}

void this_row(direction_s d){                               ❹
   printf( d.left ? "*..\n"
         : d.right ? "..*\n"
         : ".*.\n");
}

void draw_box(direction_s d){
   printf("%s:\n", (d.name ? d.name : "a box"));
   d.up               ? this_row(d) : printf("...\n");
   (!d.up && !d.down) ? this_row(d) : printf("...\n");
   d.down             ? this_row(d) : printf("...\n");
   printf("\n");
}
```

❶ Это наш первый позиционный инициализатор. Поскольку элементы `.right`, `.up` и `.down` не заданы, то им присваивается значение 0.

❷ Естественно сначала задавать имя, поэтому мы можем указать его в качестве первого инициализатора, опустив метку, – к неоднозначности это не приведет.

❸ Это крайний случай – все элементы структуры инициализируются нулями.

❹ Все, что идет после этой строки, касается отображения решетки на экране, ничего нового в этом нет.

Раньше для инициализации структуры нужно было помнить порядок ее элементов и перечислять их значения без меток, поэтому объявление `upper right` выглядело бы так:

```
direction_s upright = {NULL, 0, 1, 1, 0};
```

Эту запись невозможно понять, именно из-за таких вещей язык С не любят. Поэтому, за исключением редких ситуаций, когда порядок естественный и очевидный, не пользуйтесь синтаксисом без меток.

- Вы заметили, что при инициализации структуры `upper right` элементы перечислены не в том порядке, в каком они следуют в объявлении структуры? Жизнь слишком коротка для того, чтобы помнить порядок элементов в множествах, где он выбран произвольно, – пусть разбирается компилятор.
- Необъявленные элементы инициализируются нулями. Ни один элемент не остается неопределенным.

- Позиционные и обычные инициализаторы могут употребляться в одном предложении. В примере 10.7 казалось естественным, что имя идет вначале (и что строка "upper right" не является целым числом), поэтому даже если имя не снабжено явной меткой, объявление все равно будет корректным. Общее правило состоит в том, что компилятор продолжает с того места, на котором остановился:

```
typedef struct{
   int one;
   double two, three, four;
} n_s;

n_s justone = {10, .three=8};        // 10 без метки сопоставляется первому элементу:
                                     // .one=10
n_s threefour = {.two=8, 3, 4};      // по правилу продолжения, 3 сопоставляется
                                     // элементу, следующему за .two: .three=3 и .four=4
```

Я начал знакомство с составными литералами на примере массивов, но если принять во внимание, что структуры – это почти массивы, только с именованными элементами неодинакового размера, то становится понятно, что составные литералы годятся и для структур, что и было продемонстрировано на примере структур `upper right` и `center`. Как и раньше, перед фигурными скобками нужно поставить оператор приведения типа `(имя_типа)`.

Первый пример в `main` – непосредственное объявление, синтаксис составного инициализатора здесь не нужен; в последующих примерах с помощью составного литерала инициализируется анонимная структура, которая затем копируется в переменную `D` или передается функции.



Ваша очередь. Перепишите объявления всех структур в своем коде с использованием позиционных инициализаторов. Да-да, именно это я и хотел сказать. Старомодная инициализация, в которой не понятно, какие значения каким полям присваиваются, просто ужасна. Обратите внимание, что неказистый код вида

```
direction_s D;
D.left = 1;
D.right = 0;
D.up = 1;
D.down = 0;
```

можно переписать и так:

```
direction_s D = {.left=1, .up=1};
```

Инициализация массивов и структур нулями

Переменная, объявленная внутри функции, автоматически не обнуляется (и для переменных, которые называются автоматическими, это, пожалуй, несколько странно). Полагаю, что так сделано ради повышения быстродействия: для обнуления кадра стека, отведенного функции, требуется дополнительное время, и если функция вызывается миллион раз, а на дворе 1985 год, то эти накладные расходы могут стать ощутимыми.

Но в наши дни оставлять переменную неинициализированной значит напрашиваться на неприятности.

Простой числовой переменной можно присвоить нулевое значение в той строке, где она объявлена. Указателям, в том числе строкам, присваивайте значение NULL. Ничего сложного тут нет, а если вы забудете, то хороший компилятор предупредит о попытке использования неинициализированной переменной.

Что касается структур и массивов фиксированного размера, то, как было показано выше, компилятор обнуляет все элементы, кроме тех, которым явно присвоено значение с помощью позиционного инициализатора. Поэтому, чтобы обнулить всю структуру, достаточно задать пустой инициализатор. Ниже эта идея иллюстрируется на примере программы, которая не делает ничего полезного:

```
typedef struct {
    int la, de, da;
} ladeda_s;

int main(){
    ladeda_s emptystruct = {};
    int ll[20] = {};
}
```

Просто и красиво, правда?

А теперь о грустном: допустим, имеется массив переменной длины (которая задается с помощью переменной во время выполнения). Единственный способ инициализировать его – воспользоваться функцией memset:

```
int main(){
    int length=20;
    int ll[length];
    memset(ll, 0, 20*sizeof(int));
}
```

Так уж устроена жизнь[1].



Ваша очередь. Напишите макрос для объявления массива переменной длины и обнуления всех его элементов. На вход макросу нужно подать тип, имя и размер массива.

Для инициализации разреженных, но все же не пустых массивов можно использовать позиционные инициализаторы:

```
//По правилу продолжения, следующее объявление эквивалентно {0, 0, 1.1, 0, 0, 2.2, 3.3}:
double list1[7] = {[2]=1.1, [5]=2.2, 3.3}
```

Псевдонимы типов спешат на помощь

Позиционные инициализаторы дают новую жизнь структурам, и в оставшейся части главы мы поговорим о том, что же можно делать с помощью структур теперь,

---

[1] За это следует винить §6.7.8(3) стандарта ISO C, поскольку там четко сказано, что массивы переменной длины инициализировать запрещается. А мне так кажется, что компилятор должен был бы разобраться.

когда использовать их стало намного проще. Но сначала нужно определиться с форматом объявления. Я буду пользоваться таким форматом:

```
typedef struct newstruct_s {
  int a, b;
  double c, d;
} newstruct_s;
```

Здесь объявлен новый тип (newstruct_s), совпадающий с типом структуры (struct newstruct_s). Многие авторы выбирают разные имена для самой структуры и псевдонима ее типа (typedef), например: typedef struct _nst { ... } newstruct_s;. Это совершенно необязательно: имена структур находятся в пространстве имен, отделенном от всех прочих идентификаторов (K&R, 2-е издание, §A8.3 (стр. 213); C99 и C11 §6.2.3(1)), поэтому никаких неоднозначностей для компилятора не возникает. Я считаю, что использование одного и того же имени не мешает и человеку, зато избавляет нас от необходимости изобретать еще одно соглашение об именовании.

В стандарте POSIX имена, оканчивающиеся суффиксом _t, зарезервированы для типов, которые могут быть добавлены в стандарт в будущем. Формально стандарт C резервирует только имена int..._t и unit..._t, но в каждой новой версии появляются файлы-заголовки, в которых объявлены типы с именами, оканчивающимися на _t. Многие авторы не задумываются о конфликтах имен, которые могут возникнуть в их программах с выходом гипотетического стандарта C22, и беспечно называют свои типы именами, оканчивающимися на _t. Лично я в этой книге использую для структур имена с суффиксом _s.

Структуру этого типа можно объявить двумя способами:

```
newstruct_s ns1;
struct newstruct_s ns2;
```

Есть несколько случаев, когда необходимо использовать форму struct newstruct_s вместо newstruct_s.

- Если структура включает элемент определяемого типа. Например, в структуре связанного списка может быть поле next, являющееся указателем на другую такую же структуру:

  ```
  typedef struct newstruct_s {
     int a, b;
     double c, d;
     struct newstruct_s *next;
  } newstruct_s;
  ```

- В стандарте C11 требуется, чтобы анонимные структуры объявлялись в виде struct newstruct_s. Мы еще вернемся к этому вопросу в разделе «C без зазоров» ниже.
- Некоторые просто любят формат struct newstruct_s, и это подводит нас к вопросу о стиле.

К вопросу о стиле

Я с удивлением узнал, что многие считают, будто псевдонимы типов только запутывают программу. Вот, например, цитата из руководства по стилю программирования ядра Linux: «Как понять, что могло бы означать объявление `vps_t a;`, встречающееся в исходном коде? Но если вы видите объявление `struct virtual_container *a;`, то сразу понятно, что такое `a`». На это замечание можно привести естественное возражение: проясняет код более длинное имя – даже оканчивающееся словом `container`, – а не ключевое слово `struct` в начале.

Но у такого неприятия typedef должны же быть какие-то корни. При более пристальном исследовании обнаружились исходные файлы, в которых псевдонимы типов рекомендовалось использовать для определения единиц измерения, например:

```
typedef double inches;
typedef double meters;

inches length1;
meters length2;
```

Теперь приходится при каждом использовании типа `inches` вспоминать, что же это такое на самом деле (`unsigned int? double?`), и при этом мы даже не получаем никакой защиты от ошибок. Где-нибудь через сотню строку, написав такое присваивание:

```
length1 = length2;
```

вы уже позабыли о том, как ловко определили типы, а типичный компилятор C и не подумает предупредить вас об ошибке. Если вас заботят единицы измерения, укажите их прямо в имени переменной, и тогда ошибка станет очевидной:

```
double length1_inches, length2_meters;

// спустя 100 строчек:

length1_inches = length2_meters; // в этой строке очевидная ошибка
```

Использовать глобальные псевдонимы типов, внутреннее устройство которых должно быть известно пользователю, имеет смысл настолько же редко, насколько и прочие глобальные элементы, потому что на поиск их объявления приходится отвлекаться точно так же, как на поиск объявления переменной, а это означает, что привносимая ими структуризация сводится на нет когнитивной нагрузкой на читателя программы.

При всем при том практически в любой реальной библиотеке широко используются псевдонимы типов для глобальных структур. Например, в GSL мы встречаем typedef'ы `gsl_vector` и `gsl_matrix`, а в GLib – typedef'ы для хэшей, деревьев и прочих объектов. Даже в исходном коде Git написанной Линусом Торвальдсом системе управления версиями для ядра Linux есть несколько стратегических typedef'ов для структур.

Область видимости псевдонима типа такая же, как для любого объявления. Это означает, что можно определить typedef внутри файла, не боясь засорить пространство имен вне этого файла. Бывают даже случаи, когда разумно определить typedef внутри одной функции. Вероятно, вы заметили, что большинство встречавшихся в книге typedef'ов были локальными, то есть для поиска определения читателю достаточно просмотреть несколько предыдущих строчек, а немногие глобальные (то есть определенные в каком-то включаемом заголовке) typedef'ы тем или иным способом обернуты, так что читателю их определение вообще неинтересно. Таким образом, можно избежать когнитивной нагрузки.

Возврат нескольких значений из функции

Областью значений математической функции может быть не только одномерное пространство. Так, вполне обычны функции, значениями которых являются точки на двухмерной плоскости (x, y).

Python (и многие другие языки) позволяет возвращать несколько значений с помощью списков, например:

```
# Вернуть ширину и высоту листа бумаги одного из стандартных форматов
def width_length(papertype):
   if (papertype=="A4"):
      return [210, 297]
   if (papertype=="Letter"):
      return [216, 279]
   if (papertype=="Legal"):
      return [216, 356]

[a, b] = width_length("A4");
print("ширина=%i, высота=%i"%(a, b))
```

В C можно вернуть структуру, а стало быть, столько элементов, сколько необходимо. Именно поэтому я так восхищался возможностью создавать одноразовые структуры: сгенерировать структуру для одной конкретной функции совсем не сложно.

Но взглянем правде в глаза: C в этом отношении все еще более многословен, чем языки, имеющие специальный синтаксис для возврата списков. Однако, как показано в примере 10.8, вполне возможно ясно выразить тот факт, что функция возвращает значение из $\mathbb{R}^2$.

Пример 10.8 ❖ Если функция должна возвращать несколько значений, возвращайте структуру (papersize.c)

```
#include <stdio.h>
#include <strings.h> //strcasecmp (из POSIX)
#include <math.h> //NaN

typedef struct {
   double width, height;
} size_s;
```

```
size_s width_height(char *papertype){
   return
     !strcasecmp(papertype, "A4")     ? (size_s) {.width=210, .height=297}
   : !strcasecmp(papertype, "Letter") ? (size_s) {.width=216, .height=279}
   : !strcasecmp(papertype, "Legal")  ? (size_s) {.width=216, .height=356}
                                      : (size_s) {.width=NAN, .height=NAN};
}

int main(){
   size_s a4size = width_height("a4");
   printf("ширина=%g, высота=%g\n", a4size.width, a4size.height);
}
```



В этом примере используется конструкция *условие ? если_истина : иначе*, представляющая собой одно выражение, которое, следовательно, может встречаться сразу после `return`. Обратите внимание, как элегантно эта последовательность позволяет перечислить все случаи (в том числе и последний, соответствующий всем остальным форматам). Я люблю таким образом оформлять небольшие таблицы, но есть люди, которые находят подобный стиль отвратительным.

Альтернативный способ – использование указателей – часто встречается и не считается дурным тоном, но при этом бывает трудно понять, какие параметры входные, а какие – выходные, так что вариант с дополнительным typedef стилистически выглядит предпочтительнее:

```
// И высота, и ширина возвращаются по указателям:
void width_height(char *papertype, double *width, double *height);

// Ширина возвращается непосредственно, а высота – по указателю:
double width_height(char *papertype, double *height);
```

Извещение об ошибках

В работе [Goodliffe 2006] обсуждаются различные способы возврата из функции кода ошибки. Автор оценивает варианты довольно пессимистически.

- Иногда можно выделить специальное значение, обозначающее ошибку, например −1 для целых или NaN для чисел с плавающей точкой (однако нередки и случаи, когда допустимые возвращаемые значения занимают весь доступный диапазон).
- Можно поднимать глобальный флаг ошибки, но в 2006 году Гудлиф еще не мог рекомендовать использовать для этой цели ключевое слово `_Thread_local`, появившееся только в стандарте C11 и обеспечивающее корректную работу в многопоточных программах. Хотя в общем случае глобальный для всей программы флаг ошибки неприемлем, несколько тесно связанных между собой функций вполне могут совместно пользоваться переменной типа `_Thread_local` в области видимости файла.
- Третий вариант – «возвращать значение составного типа (кортеж), содержащее как собственно возвращаемое значение, так и код ошибки. В популярных C-подобных языках это выглядит довольно громоздко и встречается редко».

Как мы уже видели в этой главе, у возврата структур есть много достоинств, а современные версии C предлагают целый ряд средств (псевдонимы типов, позиционные инициализаторы) для борьбы с громоздкостью.



При разработке каждой новой структуры подумайте, нет ли смысла предусмотреть в ней поле `error` или `status`. Тогда при возврате такой структуры из функции у вас уже будет встроенный способ сообщить, является результат правильным или ошибочным.

В примере 10.9 уравнение из элементарного курса физики оформлено в виде функции, которая отвечает на вопрос «какую энергию приобретет идеальный объект заданной массы после свободного падения на землю в течение заданного количества секунд?» и при этом проверяет ошибки.

Я заготовил несколько макросов, потому что, согласно моим наблюдениям, многие используют макросы в C прежде всего для обработки ошибок, наверное, потому что никто не хочет, чтобы проверка ошибок отвлекала от основной задачи алгоритма.

Пример 10.9 ❖ Если нужно вернуть значение и код ошибки, воспользуйтесь структурой (errortuple.c)

```
#include <stdio.h>
#include <math.h> //NaN, pow

#define make_err_s(intype, shortname)  \                          ❶
   typedef struct { \
      intype value; \
      char const *error;  \                                       ❷
   } shortname##_err_s;

make_err_s(double, double)
make_err_s(int, int)
make_err_s(char *, string)

double_err_s free_fall_energy(double time, double mass){
   double_err_s out = {}; //initialize to all zeros.
   out.error = time < 0 ? "время отрицательно"
                        : mass < 0 ? "масса отрицательна"         ❸
                        : isnan(time) ? "время NaN"
                        : isnan(mass) ? "масса NaN"
                        : NULL;
   if (out.error) return out;                                     ❹

   double velocity = 9.8*time;
   out.value = mass*pow(velocity, 2)/2.;
   return out;
}

#define Check_err(checkme, return_val) \                          ❺
   if (checkme.error) {fprintf(stderr, "error:%s\n", checkme.error); \
                      return return_val;}

int main(){
   double notime=0, fraction=0;
```

```
    double_err_s energy = free_fall_energy(1, 1);          ❻
    Check_err(energy, 1);
    printf("Energy after one second:%g Joules\n", energy.value);
    energy = free_fall_energy(2, 1);
    Check_err(energy, 1);
    printf("Energy after two seconds:%g Joules\n", energy.value);
    energy = free_fall_energy(notime/fraction, 1);
    Check_err(energy, 1);
    printf("Energy after 0/0 seconds:%g Joules\n", energy.value);
}
```

❶ Если вам по душе идея возвращать кортеж «значение–ошибка», то понадобится отдельный тип для каждого типа возвращаемого значения. Поэтому я решил написать макрос, который позволит без труда порождать тип кортежа, соответствующий базовому типу. Чуть ниже показано его применение для генерации типов `double_err_s`, `int_err_s` и `string_err_s`. Если вы считаете этот трюк излишним, можете им не пользоваться.

❷ Почему бы не кодировать ошибки строками, а не целыми числами? Сообщения об ошибках обычно представляют собой константные строки, поэтому проблем с управлением памятью не возникнет, а в результате не нужно будет искать определение загадочного элемента перечисления. См. обсуждение в разделе «Перечисления и строки» выше.

❸ Еще одна таблица возвращаемых значений. Это типичная проверка входных параметров функции. Обратите внимание, что элемент `out.error` указывает на одну из литеральных строк. Поскольку строки не копируются, не нужно ни выделять, ни освобождать память. Чтобы подчеркнуть этот момент, я объявил указатель на `char const`.

❹ Или можно воспользоваться макросом `Stopif`, который описан в разделе «Проверка ошибок» на стр. 78: `Stopif(out.error, return out, out.error)`.

❺ Макросы для проверки возвращенного функцией значения – стандартная идиома C. Поскольку ошибка представлена строкой, макрос может вывести ее на stderr (или в журнал ошибок) непосредственно.

❻ Порядок применения вполне ожидаемый. Авторы часто сетуют, что пользователь легко может не обратить внимания на то, что функция вернула код ошибки, и в этом отношении включение выходного значения в кортеж служит неплохим напоминанием о том, что результат содержит и код ошибки, который нужно принимать во внимание.

Гибкая передача аргументов функциям

Функции могут принимать переменное число аргументов (иногда такие функции называют *вариадическими*). Самый известный пример – функция `printf`, для которой оба обращения – `printf("Hi.")` и `printf("%f%f%i\n", first, second, third)` – допустимы.

Если говорить упрощенно, то механизм функций с переменным числом аргументов достаточен для реализации `printf`, и на этом его возможности исчерпы-

ваются. Обязательно должен присутствовать первый фиксированный аргумент, и обычно ожидается, что этот первый аргумент дает возможность узнать типы или хотя бы число последующих аргументов. В примере выше первый аргумент (`"%f%f%i\n"`) говорит, что далее ожидаются еще два аргумента с плавающей точкой, а последний аргумент должен быть целым числом.

Ни о какой типобезопасности и речи нет: если вы передадите число типа `int`, например 1, вместо ожидаемого числа типа `float`, например 1.0, то результат не определен. Если функция ожидает получить три аргумента, а передано только два, то, скорее всего, дело закончится нарушением защиты памяти. Из-за подобных проблем CERT, группа по изучению безопасности программного обеспечения, считает функции с переменным числом аргументов небезопасными (степень опасности: высокая, шансы: вероятна)[1].

Мы уже видели, как можно в какой-то мере обезопасить функции с переменным числом аргументов одного и того же типа: написать макрос-обертку, который дописывает завершающий элемент в конец списка, гарантируя тем самым, что обернутая функция не получит незавершенного списка. Составной литерал проверяет также типы входных параметров и не будет компилироваться, если передан параметр не того типа.

В этом разделе мы рассмотрим еще два способа реализации функций с переменным числом аргументов, поддерживающих проверку типов. Последний способ позволяет именовать аргументы, что также до некоторой степени уменьшает вероятность ошибки. Я согласен с CERT в том, что неконтролируемое использование функций с переменным числом аргументов слишком опасно, и в своих программах пользуюсь только описанными ниже частными случаями.

Первый способ основан на том, что компилятор умеет проверять аргументы `printf`, а второй – на использовании макросов с переменным числом аргументов для подготовки входных параметров к применению синтаксиса позиционных инициализаторов в заголовках функций.

Объявление своей функции по аналогии с printf

Для начала исследуем традиционный путь и воспользуемся средствами поддержки функций с переменным числом аргументов, описанными в стандарте C89. Я останавливаюсь на этом, потому что бывают ситуации, когда макросами воспользоваться невозможно. Обычно они обусловлены психологическими, а не техническими факторами – мало найдется случаев, когда функцию с переменным числом аргументов нельзя было бы заменить макросом с переменным числом аргументов, воспользовавшись одним из рассмотренных в этой главе подходов.

Чтобы сделать функцию с переменным числом аргументов, отвечающую стандарту C89, безопасной, нам понадобится расширение, впервые предложенное в gcc, но впоследствии широко поддержанное другими компиляторами: ключевое слово `__attribute__`, открывающее доступ к различным зависящим от компилято-

---

[1] http://bit.ly/SAJTl7.

ра трюкам[1]. Оно должно находиться в строке объявления переменной, структуры или функции (поэтому если ваша функция не определена до использования, то придется ее определить).

Допустим, требуется версия функции system, принимающая аргументы так же, как printf. В примере 10.10 показана функция system_w_printf, которая принимает аргументы в духе printf, записывает их в строку и передает результат стандартной функции system. В ней используется функция vasprintf – вариант asprintf, умеющий работать с va_list. Обе эти функции описаны в стандартах BSD и GNU. Если требуется строго придерживаться стандарта C99, замените vasprintf функцией vsnprintf, являющейся аналогом snprintf (и добавьте директиву #include <stdarg.h>).

Функция main просто демонстрирует порядок применения: она берет первый аргумент из командной строки и подает его на вход команды ls.

Пример 10.10 ❖ Старомодный способ обработки списков аргументов переменной длины (olden_varargs.c)

```
#define _GNU_SOURCE //чтобы stdio.h включал vasprintf
#include <stdio.h> //printf, vasprintf
#include <stdarg.h> //va_start, va_end
#include <stdlib.h> //system, free
#include <assert.h>

int system_w_printf(char const *fmt, ...) __attribute__ \   ❶
((format (printf,1,2)));

int system_w_printf(char const *fmt, ...){
  char *cmd;
  va_list argp;
  va_start(argp, fmt);
  vasprintf(&cmd, fmt, argp);
  va_end(argp);
  int out= system(cmd);
  free(cmd);
  return out;
}

int main(int argc, char **argv){
  assert(argc == 2);                                        ❷
  return system_w_printf("ls%s", argv[1]);
}
```

[1] Если вас беспокоит, что у пользователя может оказаться компилятор, не поддерживающего ключевого слова __attribute__, то в Autotools найдутся средства утешить вас. Извлеките из архива Autoconf макрос AX_C___ATTRIBUTE__ и вставьте его в файл aclocal.m4 в каталоге проекта. Добавьте вызов AX_C___ATTRIBUTE__ в файл configure.ac, а потом с помощью препроцессора определите пустой символ __attribute__ в случае, когда Autoconf выясняет, что компилятор не поддерживает этого ключевого слова. Это делается так:

```
#include "config.h"
#ifndef HAVE___ATTRIBUTE__
#define __attribute__(...)
#endif
```

❶ Помечает эту функцию как printf-подобную, сообщая компилятору, что первый параметр – это спецификатор формата, а список дополнительных параметров начинается со второго.
❷ Сознаюсь – тут я поленился. Необернутый макрос assert следует использовать только для проверки значений, напрямую контролируемых автором, но не для данных, полученных от пользователя. Макрос, пригодный для проверки входных параметров, приведен в разделе «Проверка ошибок» выше.

Единственный плюс этого подхода, по сравнению с макросом с переменным числом аргументов, – громоздкость получения от макроса возвращенного значения. Однако версия из примера 10.11 на основе макроса короче и проще, и если компилятор поддерживает проверку типов аргументов для функций из семейства printf, то эта поддержка будет автоматически работать и здесь (без использования специфичных для gcc и Clang атрибутов).

Пример 10.11 ❖ Версия с использованием макроса, имеющая меньше зависимостей (macro_varargs.c)

```
#define _GNU_SOURCE     //чтобы stdio.h включал vasprintf
#include <stdio.h>      //printf, vasprintf
#include <stdlib.h>     //system
#include <assert.h>

#define System_w_printf(outval, ...) {\
  char *string_for_systemf; \
  asprintf(&string_for_systemf, __VA_ARGS__); \
  outval = system(string_for_systemf);  \
  free(string_for_systemf); \
}

int main(int argc, char **argv){
  assert(argc == 2);
  int out;
  System_w_printf(out, "ls%s", argv[1]);
  return out;
}
```

Необязательные и именованные аргументы

Я уже показывал, как передать функции список однотипных аргументов с помощью комбинации составного литерала и макроса с переменным числом аргументов (см. раздел «Безопасное завершение списков» выше в этой главе).

Структура во многих отношениях аналогична массиву, только позволяет хранить данные разных типов. А раз так, то можно применить ту же самую схему: написать макрос-обертку для упаковки элементов в структуру, а затем передать эту структуру функции. В примере 10.12 показано, как это делается.

Макрос собирает воедино функцию, принимающую переменное число именованных аргументов. Определение функции состоит из трех частей: одноразовой структуры, к которой никогда не обращаются по имени (однако оно все равно за-

соряет глобальное пространство имен, если функция глобальна); макроса, который вставляет свои аргументы в структуру, которая затем передается обертываемой функции, и самой обертываемой функции.

Пример 10.12 ❖ Функция, принимающая переменное число именованных аргументов, – аргументы, не заданные пользователем, будут иметь значения по умолчанию (ideal.c)

```
#include <stdio.h>

typedef struct {      ❶
  double pressure, moles, temp;
} ideal_struct;

/** Найти объем (в кубических метрах), занимаемый идеальным газом,
    который подчиняется закону V=nRT/P
Входные параметры:
давление в атмосферах (по умолчанию 1)
молярный объем вещества (по умолчанию 1)
температура в градусах Кельвина (по умолчанию температура замерзания воды = 273.15)
*/
#define ideal_pressure(...) ideal_pressure_base((ideal_struct) \  ❷
{.pressure=1, .moles=1, .temp=273.15, __VA_ARGS__})

double ideal_pressure_base(ideal_struct in){ ❸
  return 8.314 * in.moles*in.temp/in.pressure;
}

int main(){
  printf("объем с параметрами по умолчанию:%g\n", ideal_pressure() );
  printf("объем при температуре кипения:%g\n",
                ideal_pressure(.temp=373.15) ); ❹
  printf("объем двух молей вещества:%g\n", ideal_pressure(.moles=2) );
  printf("объем двух молей вещества при температуре кипения:%g\n",
                ideal_pressure(.moles=2, .temp=373.15) );
}
```

❶ Прежде всего необходимо определить структуру, содержащую выходные аргументы функции.
❷ Входные параметры макроса подставляются в определение анонимной структуры, причем аргументы, поставленные пользователем в скобках, станут позиционными инициализаторами.
❸ Сама функция принимает структуру `ideal_struct`, а не список независимых аргументов, как обычно.
❹ Пользователь задает список позиционных инициализаторов. Аргументы, не указанные явно, принимают значения по умолчанию, после чего функции `ideal_pressure_base` передается входная структура, содержащая все необходимое.

Вот как расширяется вызов функции в последней строке (не говорите пользователю, что на самом деле это макрос):

```
ideal_pressure_base((ideal_struct){.pressure=1, .moles=1, .temp=273.15,
                                                  .moles=2, .temp=373.15})
```

Общее правило состоит в том, что если некоторое поле инициализируется несколько раз, то предпочтение отдается последней инициализации. Поэтому поле .pressure будет иметь значение по умолчанию, тогда как остальные два входных параметра – значения, заданные пользователем.



В Clang повторная инициализация полей moles и temp приводит к предупреждению, если задан флаг -Wall, поскольку авторы компилятора сочли, что двойная инициализация скорее свидетельствует об ошибке, чем о сознательном задании значений по умолчанию. Чтобы отключить это предупреждение, добавьте флаг -Wnoinitializer-overrides. gcc считает такую ситуацию ошибкой, только если явно задан флаг -Wextra warnings; чтобы в таком режиме отключить это конкретное предупреждение, добавьте флаги -Wextra -Woverride-init.



Ваша очередь. В данном случае одноразовая структура, быть может, и не такая уж одноразовая, потому что имеет смысл применять формулу в различных направлениях:

- давление = 8.314 моля * температура / объем;
- моль = давление * объем / (8.314 температура);
- температура = давление * объем /(8.314 моля).

Переделайте структуру, добавив в нее поле объема, и, пользуясь одной и той же структурой, напишите функции для этих дополнительных уравнений.

Затем с помощью этих функций напишите унифицированную функцию, которая принимает структуру с тремя заполненными полями из четырех (четвертое может содержать NaN, или, если хотите, можете добавить в структуру поле what_to_solve) и, применяя нужное уравнение, заполняет четвертое поле.

Теперь, когда все аргументы необязательны, вы можете вернуться к этой функции через полгода и добавить новый аргумент, не «поломав» программ, в которых она используется. Можно начать с простейшей работоспособной функции и по мере необходимости добавлять новые возможности. Однако не стоит забывать об уроках языков, в которых эта возможность была с самого начала: очень легко увлечься и начать писать функции с десятками параметров, каждый из которых необходим для обработки каких-то необычных случаев.

Доведение до ума бестолковой функции

До сих пор мы демонстрировали простые конструкции, не требующие значительных усилий от программиста, однако короткие примеры не могут охватить все приемы, применяемые для объединения разрозненных частей в полезную и надежную программу для решения реальной задачи. Поэтому в дальнейшем примеры будут усложняться и учитывать более реалистичные условия.

В примере 10.13 приведена бестолковая и неприятная функция. В случае ссуды, погашаемой в рассрочку, ежемесячные платежи фиксированы, но часть ссуды, уходящая на погашение процентов, в начале (когда заемщик еще много должен) гораздо выше и уменьшается до нуля в конце срока. Вычисления довольно утомительны (особенно если добавляется возможность вносить ежемесячные дополнительные платежи для погашения основной суммы или досрочно погасить всю ссуду), так что читателя, который захочет пропустить детали, можно извинить.

Интерес для нас представляет интерфейс, в котором 10 параметров перечисляются в некоем произвольном порядке. Применение этой функции для финансовых расчетов сопряжено с трудностями и чревато ошибками.

Таким образом, amortize – яркий пример многочисленных унаследованных функций, написанных на С за долгие годы. Панк-роком она является только в том смысле, что демонстрирует полнейшее презрение к публике. Поэтому, следуя стилю глянцевых журналов, мы в этом разделе нарядим эту функцию в яркую обертку. Если это унаследованный код, то мы не вправе изменять интерфейс самой функции (потому что от него могут зависеть другие программы), поэтому в дополнение к процедуре генерации именованных необязательных параметров, использованной в примере с уравнением идеального газа, нам понадобится промежуточная функция для наведения моста между результатом работы макроса и фиксированным набором входных аргументов унаследованной функции.

Пример 10.13 ❖ Трудная для использования функция, у которой слишком много входных аргументов и нет проверки ошибок (amortize.c)

```
#include <math.h> //pow
#include <stdio.h>
#include "amortize.h"

double amortize(double amt, double rate, double inflation, int months,
                int selloff_month, double extra_payoff, int verbose,
                double *interest_pv, double *duration,
                double *monthly_payment){
   double total_interest = 0;
   *interest_pv = 0;
   double mrate = rate/1200;

   // Месячный платеж фиксирован, но доля, уходящая на выплату процентов,
   // меняется.
   *monthly_payment = amt * mrate/(1-pow(1+mrate, -months)) + extra_payoff;
   if (verbose) printf("Ваш общий ежемесячный платеж:%g\n\n",
                       *monthly_payment);
   int end_month = (selloff_month && selloff_month < months )
                        ? selloff_month
                        : months;
   if (verbose)
      printf("yr/mon\t Princ.\t\tInt.\t| PV Princ.\t PV Int.\t Ratio\n");
   int m;
   for (m=0; m < end_month && amt > 0; m++){
      double interest_payment = amt*mrate;
      double principal_payment = *monthly_payment - interest_payment;
      if (amt <= 0)
          principal_payment =
          interest_payment = 0;
      amt -= principal_payment;
      double deflator = pow(1 + inflation/100, -m/12.);
      *interest_pv += interest_payment * deflator;
      total_interest += interest_payment;
      if (verbose) printf("%i/%i\t%7.2f\t\t%7.2f\t|%7.2f\t%7.2f\t%7.2f\n",
```

```
            m/12, m-12*(m/12)+1, principal_payment, interest_payment,
            principal_payment*deflator, interest_payment*deflator,
            principal_payment/(principal_payment+interest_payment)*100);
    }
    *duration = m/12.;
    return total_interest;
}
```

В примерах 10.14 и 10.15 организуется удобный интерфейс к этой функции. Файл-заголовок содержит главным образом документацию в формате Doxygen, потому что входных параметров очень много, и было бы безумием оставить их недокументированными, а кроме того, пользователю необходимо сообщить, каковы будут значения по умолчанию для параметров, не заданных явно.

Пример 10.14 ❖ Файл-заголовок, включающий в основном документацию, а также макрос и заголовок промежуточной функции (amortize.h)

```
double amortize(double amt, double rate, double inflation, int months,
         int selloff_month, double extra_payoff, int verbose,
         double *interest_pv, double *duration, double *monthly_payment);

typedef struct {
   double amount, years, rate, selloff_year, extra_payoff, inflation;      ❶
   int months, selloff_month;
   _Bool show_table;
   double interest, interest_pv, monthly_payment, years_to_payoff;
   char *error;
} amortization_s;

/** Вычислить скорректированную с учетом инфляции величину процентов,        ❷
  уплачиваемых в течение срока предоставления ссуды, например ипотечного
  кредита.

\li \c amount  Размер ссуды в долларах. Значения по умолчанию нет--если
               не задан, печатается сообщение об ошибке и возвращаются нули.
\li \c months  Количество месяцев, на которое выдана ссуда. По умолчанию:
               нуль, однако см. параметр years.
\li \c years Если параметр months не задан, то можно задать количество лет.
             Например, 10.5=десять лет и шесть месяцев. По умолчанию: 30
             (типичный срок ипотечного кредитования в США).
\li \c rate Процентная ставка в годовом исчислении. По умолчанию: 4.5
            (то есть 4.5%), типичная ставка для рынка ипотечного кредитования
            США в 2012 году.
\li \c inflation Процентный уровень инфляции за год, используется для
                 расчета стоимости денег. По умолчанию: 0, то есть поправка
                 на инфляцию отсутствует. В последние несколько десятков
                 лет для США характерно значение в районе 3.
\li \c selloff_month В этот месяц ссуда может быть выплачена полностью
                     (например, если вы перепродали дом). По умолчанию: 0
                     (то есть досрочное погашение не предусмотрено).
\li \c selloff_year Если selloff_month==0 и этот параметр положителен, то
                    он определяет год досрочного погашения. По умолчанию: 0
                    (то есть досрочное погашение не предусмотрено).
```

```
\li \c extra_payoff Дополнительный ежемесячный платеж для погашения основной
                     суммы долга. По умолчанию: 0.
\li \c show_table  Если отличен от нуля, то выводится таблица платежей. Если
                   равен 0, то не выводится ничего (просто возвращается
                   общая сумма процентов). По умолчанию: 1.

Все параметры, кроме \c extra_payoff и \c inflation, должны быть неотрицательны.

\return структура \c amortization_s, в которой заполнены все вышеперечисленные поля,
а также:

\li \c interest Общая сумма, уплачиваемая в качестве процентов.
\li \c interest_pv Общая сумма уплачиваемых процентов с поправкой текущей
                   стоимости на инфляцию.
\li \c monthly_payment Фиксированный ежемесячный платеж (в случае ипотечного
                       кредита к нему добавляются налоги и проценты).
\li \c years_to_payoff Обычно срок предоставления ссуды или дата досрочного
                       погашения, но если вы уплачиваете больше
                       фиксированного платежа, то ссуда будет погашена
                       раньше.
\li \c error Если <tt>error != NULL</tt>, значит, произошла какая-то ошибка
и результаты недостоверны.
*/

#define amortization(...) amortize_prep((amortization_s){.show_table=1, \
                                                   __VA_ARGS__})      ❸

amortization_s amortize_prep(amortization_s in);
```

❶ Структура, с помощью которой макрос передает данные промежуточной функции. Она должна находиться в той же области видимости, что сам макрос и промежуточная функция. Некоторые поля не соответствуют параметрам функции `amortize`, но упрощают работу пользователя. Часть полей заполняется функцией.

❷ Документация в формате Doxygen. Хорошо, когда документация занимает большую часть интерфейсного файла. Обратите внимание, что для каждого входного параметра указано значение по умолчанию.

❸ Этот макрос помещает введенные пользователем данные – скажем, `amortization(.amount=2e6, .rate=3.0)` – в позиционный инициализатор структуры `amortization_s`. Мы задали здесь значение по умолчанию для `show_table`, потому что иначе нельзя было бы отличить случай, когда пользователь явно установил `.show_table=0`, от случая, когда параметр `.show_table` вообще опущен. Таким образом, когда требуется задать отличное от нуля значение по умолчанию для параметра, который допускает нулевое значение, то нужно использовать такую форму.

Как и раньше, явственно видны три составные части механизма именованных аргументов: typedef для структуры, макрос, который принимает именованные элементы и заполняет структуру, и функция, которая принимает эту структуру

в качестве параметра. Однако в данном случае вызывается промежуточная функция, расположенная между макросом и обертываемой функцией. Ее объявление находится в заголовке, а определение показано в примере 10.15.

Пример 10.15 ❖ Закрытая часть интерфейса (amort_interface.c)

```
#include "stopif.h"
#include <stdio.h>
#include "amortize.h"

amortization_s amortize_prep(amortization_s in){          ❶
   Stopif(!in.amount || in.amount < 0 || in.rate < 0      ❷
         || in.months < 0 || in.years < 0 || in.selloff_month < 0
         || in.selloff_year < 0,
         return (amortization_s){.error="Недопустимые входные данные"},
            "Недопустимые входные данные. Возвращаются нули.");

   int months = in.months;
   if (!months){
      if (in.years) months = in.years * 12;
      else          months = 12 * 30; // ссуда по ипотечному кредиту
   }
   int selloff_month = in.selloff_month;
   if (!selloff_month && in.selloff_year)
      selloff_month = in.selloff_year * 12;

   amortization_s out = in;
   out.rate = in.rate ? in.rate : 4.5;                    ❸
   out.interest = amortize(in.amount, out.rate, in.inflation,
      months, selloff_month, in.extra_payoff, in.show_table,
      &(out.interest_pv), &(out.years_to_payoff), &(out.monthly_payment));
   return out;
}
```

❶ Это промежуточная функция, которая должна была бы являться частью amortize: она устанавливает разумные нетривиальные умолчания и проверяет, нет ли ошибок во входных параметрах. После этого amortize может сразу заняться своим делом, поскольку вся подготовительная работа уже произведена.

❷ Обсуждение макроса Stopif см. в разделе «Проверка ошибок» на стр. 78. Как и там, проверка в этой строке обусловлена скорее желанием предотвратить нарушения защиты памяти и удостовериться в разумности параметров, чем стремлением автоматизировать проверку всех возможных ошибок.

❸ Поскольку это простая константа, мы могли бы установить уровень инфляции и в макросе amortization вместе со значением по умолчанию для show_table. Решать вам.

Основное назначение промежуточной функции – получить одну структуру и вызвать функцию amortize, передав ей в качестве параметров отдельные поля структуры, поскольку изменять интерфейс самой функции amortize мы не вправе. Но уж коль скоро мы завели функцию, предназначенную для подготовки вход-

ных параметров, то почему бы не проверить ошибки и не установить значения по умолчанию? Например, можно разрешить пользователю задавать временные промежутки в месяцах или в годах и возвращать из промежуточной функции код ошибки, если входные параметры выходят за границы допустимого диапазона или неразумны.

Для подобных функций значения по умолчанию особенно важны, потому что большинство пользователей не знает (и не интересуется), каков разумный уровень инфляции. Если компьютер способен предложить пользователю знания о предметной области, которыми тот не располагает, и сделать это ненавязчиво, с помощью значения по умолчанию, которое легко переопределить, то вряд ли пользователь будет расстроен.

Функция amortize возвращает несколько значений. Как сказано в разделе «Возврат нескольких значений из функции», вместо того чтобы возвращать одно значение напрямую, а остальные – по указателям, удобно собрать все возвращаемые значения в одну структуру. Но чтобы комбинация позиционных инициализаторов и макросов с переменным числом аргументов заработала, нам уже пришлось завести промежуточную структуру; так почему бы не объединить две структуры в одну? В результате мы получаем на выходе структуру, в которой сохранены также все входные параметры.

Организовав такой интерфейс, мы получим хорошо документированную, простую в использовании функцию с проверкой ошибок, и с помощью приведенной в примере 10.16 программы сможем без труда анализировать разнообразные ситуации. Эта программа собирается из файлов *amortize.c* и *amort_interface.c*, причем в последнем используется функция pow из математической библиотеки, так что makefile выглядит следующим образом:

```
P=amort_use
objects=amort_interface.o amortize.o
CFLAGS=-g -Wall -O3 #the usual
LDLIBS=-lm
CC=c99
$(P):$(objects)
```

Пример 10.16 ❖ Сейчас мы можем воспользоваться макросом и функцией для расчета процентов по ссуде для моделирования различных ситуаций (amort_use.c)

```
#include <stdio.h>
#include "amortize.h"

int main(){
    printf("Типичная ссуда:\n");
    amortization_s nopayments = amortization(.amount=200000, .inflation=3);
    printf("Вы спустили $%g в унитаз, текущая стоимость займа $%g.\n",
           nopayments.interest, nopayments.interest_pv);

    amortization_s a_hundred = amortization(.amount=200000, .inflation=3,
                                            .show_table=0, .extra_payoff=100);
    printf("Выплачивая лишние $100 в месяц, вы потеряете только $%g (PV), "
           "и весь долг будет погашен через%g лет.\n",
```

```
        a_hundred.interest_pv, a_hundred.years_to_payoff);

    printf("Погасив ссуду через десять лет, вы уплатите $%g "
           "в виде процентов (PV).\n",
           amortization(.amount=200000, .inflation=3,
                        .show_table=0, .selloff_year=10).interest_pv);  ❶
}
```

❶ Функция amortization возвращает структуру, и в первых двух случаях мы присвоили ей имя и использовали поля именованной структуры. Но если промежуточная именованная переменная не нужна, то можно ее и не заводить. В этой строчке мы извлекаем из возвращенной структуры единственное поле. Если бы функция возвращала блок памяти, выделенный с помощью malloc, то такой трюк не прошел бы, потому что переменную необходимо назвать, чтобы потом передать функции освобождения памяти. Однако в этой главе мы всюду передаем структуры, а не указатели на структуры.

Обертывание исходной функции потребовало много строк кода, но стереотипная структура и макросы для подготовки именованных аргументов занимают лишь малую их часть. Остальное – документация и предварительная обработка входных параметров, которую в любом случае имело смысл добавить. В итоге из функции с почти непригодным интерфейсом мы получили удобную для использования функцию, каким и должен быть кредитный калькулятор.

Указатель на void и структура, на которую он указывает

Этот раздел посвящен реализации обобщенных процедур и структур. В одном из рассматриваемых здесь примеров показано, как применить некоторую функцию к каждому файлу в иерархии каталогов; это может быть вывод имен файлов на экран, поиск строки и вообще все, что придет на ум пользователю. В другом примере мы воспользуемся структурой хэша из библиотеки GLib для подсчета количества отдельных символов, встречающихся в файле. Для этого придется ассоциировать ключ в виде символа Unicode с целочисленным значением. Разумеется, структура хэша в GLib допускает любые типы ключей и значений, так что подсчет символов Unicode – это пример применения обобщенного контейнера.

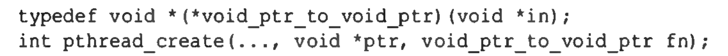
Этой гибкостью мы обязаны указателю на void, который может указывать на все, что угодно. Функции хэширования и функции обработки каталогов безразлично, на что именно ведет указатель, они просто передают значения дальше. Забота о типобезопасности ложится на программиста, но структуры помогают обеспечить ее и упрощают написание и использование обобщенных процедур.

Функции с обобщенными входными параметрами

Функцией обратного вызова называется функция, которая передается другой функции, чтобы та ее вызывала в подходящий момент. В данном случае нам нужно

рекурсивно обойти каталог и что-то сделать с каждым файлом. Для этой цели мы передаем процедуре обхода каталога функцию обратного вызова, применяемую к каждому файлу.

Проблема изображена на рис. 10.1. Когда функция вызывается напрямую, компилятор знает типы фактических данных и типы параметров функции; если они не совпадают, то компилятор об этом сообщит. Но обобщенная процедура не должна диктовать формат функции или используемых в ней данных. В разделе «Библиотека pthread» на стр. 307 рассматривается функция `pthread_create`, которую (если опустить не относящиеся к делу детали) можно было бы объявить в таком виде:



```
typedef void *(*void_ptr_to_void_ptr)(void *in);
int pthread_create(..., void *ptr, void_ptr_to_void_ptr fn);
```

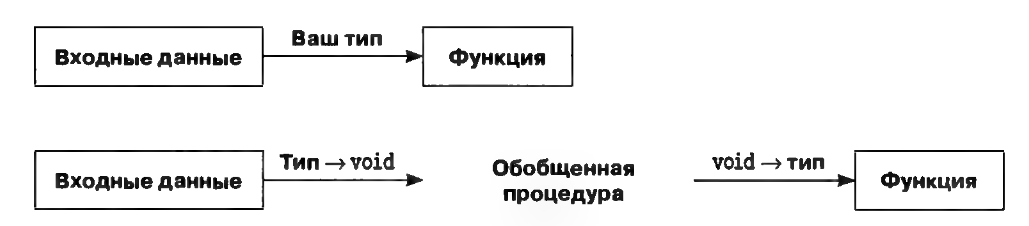


Рис. 10.1. Сравнение вызова функции напрямую и вызова из обобщенной процедуры

Если вызвать `pthread_create(..., indata, myfunc)`, то информация о типе `indata` будет потеряна, потому что было выполнено приведение к указателю на void. Где-то внутри `pthread_create` производится обращение вида `myfunc(indata)`. Если переменная `indata` имеет тип `double*`, а `myfunc` принимает `char*`, то произойдет катастрофа, которую компилятор не в силах предотвратить.

В примере 10.17 приведен файл-заголовок, необходимый для реализации функции, которая применяет некоторые функции к каждому каталогу и файлу внутри каталога. Включена документация в формате Doxygen, описывающая, что должна делать функция `process_dir`. Как и полагается, длина документации примерно совпадает с длиной самого кода.

Пример 10.17 ❖ Файл-заголовок для обобщенной функции обхода каталога (process_dir.h)

```
struct filestruct;
typedef void (*level_fn)(struct filestruct path);

typedef struct filestruct{
   char *name, *fullname;
   level_fn directory_action, file_action;
   int depth, error;
   void *data;
} filestruct;                                                    ❶

/** Получить содержимое заданного каталога, применить к каждому файлу
```

```
    функцию \c file_action, а к каждому каталогу - функцию \c dir_action и рекурсивно
    посетить каталог. Отметим, что таким образом реализуется обход в глубину.

    Пользовательские функции принимают структуру \c filestruct. Отметим, что
    в ней имеется поле \c error, в которое можно записать 1 для индикации ошибки.

    Входными параметрами являются позиционные инициализаторы. Допустимы
    следующие параметры:

    \li \c .name Имя текущего файла или каталога
    \li \c .fullname Путь к текущему файлу или каталогу
    \li \c .directory_action Функция, принимающая \c filestruct.
            При вызове этой функции ей передается правильно заполненная
            структура \c filestruct для каталога (она вызывается до
            обработки каких-либо файлов из этого каталога).
    \li \c .file_action Аналогична \c directory_action, но вызывается для
            каждого файла, а не каталога.
    \li \c .data Указатель на void, передаваемый пользовательским функциям.
    \return 0=OK, в противном случае количество каталогов, при обработке
            которых возникала ошибка, плюс количество ошибок, возвращенных
            пользовательскими функциями.

    Пример использования:
\code
    void dirp(filestruct in){ printf("Каталог: <%s>\n", in.name); }
    void filep(filestruct in){ printf("Файл:%s\n", in.name); }

    // Вывести список всех файлов в текущем каталоге, исключая каталоги:
    process_dir(.file_action=filep);

    // Показать все, что находится в домашнем каталоге:
    process_dir(.name="/home/b", .file_action=filep,
                .directory_action=dirp);
\endcode
*/
#define process_dir(...) process_dir_r((filestruct){__VA_ARGS__})        ❷
int process_dir_r(filestruct level);                                      ❸
```

❶ И снова они: три части функции, принимающей именованные аргументы. Но даже без этого приема эта структура необходима для обеспечения типобезопасности при передаче указателей на void.

❷ Макрос, который копирует позиционные инициализаторы, полученные от пользователя, в структуру составного литерала.

❸ Функция, которая принимает структуру, подготовленную макросом `process_dir`. Пользователи не должны вызывать ее напрямую.

Возвращаясь к рис. 10.1, мы видим, что этот заголовок уже содержит частичное решение проблемы типобезопасности: определить тип `filestruct` и потребовать, чтобы функция обратного вызова принимала структуру такого типа. В конце структуры по-прежнему «закопан» указатель на void. Мы могли бы вынести его наружу:

```
typedef void (*level_fn)(struct filestruct path, void *indata);
```

Но поскольку мы определяем одноразовую структуру только для поддержки функции process_dir, то можно и оставить указатель на void в ней. Далее, имея структуру, ассоциированную с функцией process_dir, мы можем воспользоваться ей для написания макроса, который преобразует позиционные инициализаторы в параметры функции, как в разделе «Необязательные и именованные аргументы» выше. Структуры вообще упрощают жизнь.

В примере 10.18 показано использование функции process_dir – части до и после облака на рис. 10.1. Функции обратного вызова просты: они всего лишь печатают отступы и имя каталога или файла. Тут даже нет никакого отступления от типобезопасности, потому что входным параметром обеих функций обратного вызова является структура вполне определенного типа.

Ниже приведен пример вывода для каталога, содержащего два файла и подкаталог *cfiles*, в котором находятся еще три файла:

```
Дерево для sample_dir:
├ cfiles
└──┐
   │ c.c
   │ a.c
   │ b.c
│ a_file
│ another_file
```

Пример 10.18 ❖ Программа вывода древовидной структуры текущего каталога (show_tree.c)

```
#include <stdio.h>
#include "process_dir.h"

void print_dir(filestruct in){
  for (int i=0; i< in.depth-1; i++) printf("   ");
  printf("├%s\n", in.name);
  for (int i=0; i< in.depth-1; i++) printf("   ");
  printf("└──┐\n");
}

void print_file(filestruct in){
  for (int i=0; i< in.depth; i++) printf("   ");
  printf("│%s\n", in.name);
}

int main(int argc, char **argv){
  char *start = (argc>1) ? argv[1] : ".";
  printf("Дерево для%s:\n", start ? start : "текущего каталога");
  process_dir(.name=start, .file_action=print_file,
              .directory_action=print_dir);
}
```

Как видите, main передает функции print_dir и print_file в process_dir в надежде, что process_dir вызовет их в нужное время с нужными аргументами.

Сама функция process_dir приведена в примере 10.19. Основная часть ее работы посвящена созданию структуры, описывающей файл или каталог, обрабатываемый в данный момент. Переданный ей каталог открывается с помощью функции opendir. Затем каждое обращение к readdir возвращает очередной элемент каталога – файл, подкаталог, ссылку и т. п. В структуру filestruct записывается информация о текущем элементе. Эта структура передается той иной функции обратного вызова в зависимости от того, что мы сейчас обрабатываем: файл или каталог. Если это каталог, то далее эта же функция вызывается для него рекурсивно.

Пример 10.19 ❖ Рекурсивно обходим каталог и применяем функцию file_action к каждому обнаруженному в нем файлу и функцию directory_action – к каждому обнаруженному каталогу (process_dir.c)

```
#include "process_dir.h"
#include <dirent.h> //struct dirent
#include <stdlib.h> //free

int process_dir_r(filestruct level){
   if (!level.fullname){
     if (level.name) level.fullname=level.name;
     else level.fullname=".";
   }
   int errct=0;

   DIR *current=opendir(level.fullname);                               ❶
   if (!current) return 1;
   struct dirent *entry;
   while((entry=readdir(current))) {
     if (entry->d_name[0]=='.') continue;
     filestruct next_level = level;                                    ❷
     next_level.name = entry->d_name;
     asprintf(&next_level.fullname, "%s/%s", level.fullname,
              entry->d_name);

     if (entry->d_type==DT_DIR){
        next_level.depth ++;
        if (level.directory_action) level.directory_action(next_level); ❸
        errct+= process_dir_r(next_level);
     }
     else if (entry->d_type==DT_REG && level.file_action){
        level.file_action(next_level);                                 ❹
        errct+= next_level.error;
     }
     free(next_level.fullname);                                        ❺
   }
   closedir(current);
   return errct;
}
```

❶ Функции opendir, readdir и closedir определены в стандарте POSIX.

❷ Для каждого элемента каталога создаем копию входной структуры filestruct и записываем в нее информацию о текущем элементе.

❸ Вызываем функцию обработки каталога, передавая ей новую структуру. Рекурсивно обрабатываем подкаталог.
❹ Вызываем функцию обработки файла, передавая ей новую структуру.
❺ Структуры `filestruct`, создаваемые на каждом шаге, – не указатели на память, выделенную с помощью `malloc`, поэтому никакой код для управления памятью не нужен. Однако `asprintf` неявно выделяет память для поля `fullname`, и эту память затем нужно освободить во избежание утечки.

Описанная стратегия позволила успешно реализовать необходимую инкапсуляцию: функции печати ничего не знают о средствах POSIX для обработки каталогов, а *process_dir.c* не в курсе того, что делают функции. И благодаря специализированной для функции структуре данных удалось сделать поток управления естественным.

Обобщенные структуры

Связанные списки, хэши, деревья и аналогичные структуры данных применяются в самых разных ситуациях, поэтому разумно сделать так, чтобы они манипулировали указателями на void, а пользователь проверял типы на входе и на выходе.

В этом разделе мы рассмотрим стандартный пример из учебника: хэш для подсчета частот вхождения символов. Хэш – это контейнер, в котором хранятся пары ключ/значение, его задача – обеспечить быстрый поиск значения по ключу.

Прежде чем приступать к обработке файлов в каталоге, необходимо настроить обобщенный хэш, имеющийся в GLib, под нужды программы; ключом у нас будет символ Unicode, а значением – целое число. Сконфигурировав этот компонент (что само по себе дает неплохой пример работы с обратными вызовами), уже легко будет реализовать функции обратного вызова для обхода файловой системы.

Как мы увидим, функции `equal_chars` и `printone` будут использоваться в роли обратных вызовов из хэша, то есть хэш будет вызывать их, передавая указатели на void. Поэтому в первых строчках этих функций объявляются переменные нужного типа и производится приведение указателя на void к указателю на конкретный тип.

В примере 10.20 показан заголовок, показывающий, что из кода в примере 10.21 предназначено для пользователей.

Пример 10.20 ❖ Заголовок для unictr.c (unictr.h)

```
#include <glib.h>

void hash_a_character(gunichar uc, GHashTable *hash);
void printone(void *key_in, void *val_in, void *xx);
GHashTable *new_unicode_counting_hash();
```

Пример 10.21 ❖ Функции для работы с хэшем, в котором ключом является символ Unicode, а значением – целочисленный счетчик (unictr.c)

```
#include "string_utilities.h"
#include "process_dir.h"
#include "unictr.h"
```

```
#include <glib.h>
#include <stdlib.h> //calloc, malloc

typedef struct {                                                          ❶
   int count;
} count_s;

void hash_a_character(gunichar uc, GHashTable *hash){
   count_s *ct = g_hash_table_lookup(hash, &uc);
   if (!ct){
      ct = calloc(1, sizeof(count_s));
      gunichar *newchar = malloc(sizeof(gunichar));
      *newchar = uc;
      g_hash_table_insert(hash, newchar, ct);
   }
   ct->count++;
}

void printone(void *key_in, void *val_in, void *ignored){                 ❷
   gunichar const *key= key_in;                                           ❸
   count_s const *val= val_in;
   char utf8[7];
   utf8[g_unichar_to_utf8(*key, utf8)]='\0';                              ❹
   printf("%s\t%i\n", utf8, val->count);
}

static gboolean equal_chars(void const * a_in, void const * b_in){       ❺
   const gunichar *a= a_in;                                               ❻
   const gunichar *b= b_in;
   return (*a==*b);
}

GHashTable *new_unicode_counting_hash(){
   return g_hash_table_new(g_str_hash, equal_chars);
}
```

❶ Да, это действительно структура, содержащая всего одно целое. В один прекрасный день она, возможно, спасет вам жизнь.

❷ Эта функция будет обратно вызываться из `g_hash_table_foreach`, поэтому она принимает указатели на void для ключа и значения и еще один указатель, который в функции не используется.

❸ Если функция принимает указатель на void, то в самом начале необходимо завести переменную правильного типа, к которому приводится этот указатель. Не откладывайте это на потом – сделайте сразу же, когда можно проверить, что приведение типа прошло успешно.

❹ Шести символов (`char`) достаточно для представления любого символа Unicode в кодировке UTF-8. Еще один байт добавлен для завершающего нуля, так что 7 байт нам всегда хватит.

❺ Поскольку в хэше могут храниться ключи и значения любого типа, GLib просит предоставить функцию сравнения двух ключей на равенства. Впоследствии функция `new_unicode_counting_hash` передаст ее функции создания хэша.

❻ Я уже говорил, что в первой строке функции, принимающей указатель на void, необходимо присвоить этот указатель переменной нужного типа. После этого вы снова оказываетесь в типобезопасной зоне.

Итак, мы имеем функции для поддержки хэша, содержащего символы Unicode. А в примере 10.22 показано, как они используются, и демонстрируется применение рассмотренной выше функции process_dir для подсчета всех символов в файлах каталога, представленных в кодировке UTF-8.

Поскольку используется та же функция process_dir, что и раньше, обобщенная процедура не должна вызывать вопросов. Функция обратного вызова для обработки одного файла hash_a_file принимает параметр типа filestruct, но внутри этой структуры скрыт указатель на void. Наши функции используют этот указатель как указатель на структуру хэша из GLib. Поэтому первым делом hash_a_file приводит указатель на void к типу указателя на эту структуру, восстанавливая тем самым типобезопасность.

Каждый компонент можно отлаживать изолированно, нужно лишь знать, что и когда подается на вход. Но можно проследить за передачей хэша из одного компонента в другой и убедиться, что он действительно передается process_dir в поле .data входной структуры filestruct, что затем hash_a_file приводит .data обратно к типу GHashTable и что далее он передается функции hash_a_character, которая увеличивает связанный с символом счетчик или добавляет новый символ в хэш. После этого g_hash_table_foreach с помощью обратного вызова printone распечатывает все элементы хэша.

Пример 10.22 ❖ Счетчик символов; используется так: charct your_dir | sort -k 2 -n (charct.c)

```
#define _GNU_SOURCE             //чтобы stdio.h определял asprintf
#include "string_utilities.h"   //string_from_file
#include "process_dir.h"
#include "unictr.h"
#include <glib.h>
#include <stdlib.h>             //free

void hash_a_file(filestruct path){
    GHashTable *hash = path.data;                              ❶
    char *sf = string_from_file(path.fullname);
    if (!sf) return;
    char *sf_copy = sf;
    if (g_utf8_validate(sf, -1, NULL)){
        for (gunichar uc; (uc = g_utf8_get_char(sf))!='\0';    ❷
             sf = g_utf8_next_char(sf))
                hash_a_character(uc, hash);
    }
    free(sf_copy);
}

int main(int argc, char **argv){
    GHashTable *hash;
```

```
    hash = new_unicode_counting_hash();
    char *start=NULL;
    if (argc>1) asprintf(&start, "%s", argv[1]);
    printf("Хешируется%s\n", start ? start : "текущий каталог");
    process_dir(.name=start, .file_action=hash_a_file, .data=hash);
    g_hash_table_foreach(hash, printone, NULL);
}
```

❶ Напомним, что в структуре filestruct есть поле data, содержащее указатель на void. Поэтому в первой строке этой функции, естественно, объявляется переменная нужного типа, которой присваивается переданный указатель на void.

❷ В кодировке UTF-8 длина символа переменна, поэтому нужна специальная функция для получения текущего символа или перехода к следующему символу в строке.

Я отношу себя к недотёпам, способным сделать ошибку в любом месте, где это возможно, и все же я редко помещал структуру неподходящего типа в список, дерево и т. п. (вообще не припоминаю такого случая!). Вот какие правила обеспечения типобезопасности я для себя выработал.

- Если есть два списка, в которых хранятся указатели void: active_groups и persons – то мне совершенно очевидно, что в строке вида g_list_append(active_groups, next_person) делается попытка поместить структуру не в тот список, и никакая помощь от компилятора мне в этом случае не нужна. Поэтому первый секрет успеха – придумывать такие имена, при которых нелепость действия видна с первого взгляда.
- Размещать обе части, показанные на рис. 10.1, как можно ближе друг к другу, так чтобы, изменив одну, вы легко могли изменить и другую.
- Я уже не раз говорил, что в начале любой функции, принимающей указатель на void, нужно скопировать этот указатель в переменную конкретного типа, выполнив тем самым приведение типов, как это сделано в printone и equal_chars. Делая это прямо «на границе», вы повышаете шансы своевременно выполнить приведение, а после того как это сделано, проблему типобезопасности можно считать решенной.
- Ассоциирование специально созданной структуры с конкретным применением обобщенной процедуры или структуры – весьма разумная мысль.
  - Если такой специальной структуры нет, то при любом изменении типа входного аргумента придется найти все случаи приведения указателя на void к указателю на старый тип и заменить в них старый тип новым. Компилятор здесь ничем не поможет. Если же вы передаете данные в специальной структуре, то нужно будет всего лишь изменить определение этой структуры.
  - Продолжая эту мысль: если вы обнаружили, что функции обратного вызова нужно передать дополнительную информацию – а весьма вероятно, что так оно будет, – то надо будет только добавить еще одно поле в определение структуры.

– Может показаться, что передача одного числа не заслуживает заведения специальной структуры, но это очень опасное заблуждение. Допустим, имеется обобщенная процедура, которая принимает функцию обратного вызова и указатель на void, который следует этой функции передать. Выглядит это так:

```
void callback (void *voidin){
   double *input = voidin;
   ...
}

int i=23;
generic_procedure(callback, &i);
```

Вы заметили, что в этом невинном, на первый взгляд, коде таится беда? Какой бы комбинацией бит ни было представлено число 23 типа int, можете не сомневаться, что после чтения его как double в функции callback ничего похожего на 23 вы не получите. Объявление новой структуры, может, и напоминает бюрократию, то помогает предотвратить эту естественную ошибку:

```
typedef struct {
   int level;
} one_lonely_integer;
```

– Мне кажется, что с точки зрения восприятия программы удобно знать, что существует единственный тип, используемый для всех схожих задач в данном участке кода. Выполняя приведение к типу, специально определенному для конкретной ситуации, я знаю, что поступаю правильно; нет места тревожным сомнениям, к какому типу приводить: char *, char ** или, может быть, к wchar_t *.

В этой главе мы рассмотрели много случаев, когда передача структуры в функцию и возврат структуры из функции улучшают программу. С помощью хорошего макроса входную структуру можно заполнить значениями по умолчанию и реализовать механизм именованных параметров. Выходную структуру можно построить на лету с помощью составного литерала. Для копирования структуры (как в случае рекурсии) достаточно простого знака равенства. Возврат пустой структуры – тривиальный случай применения позиционного инициализатора, в котором не задано ни одно поле. А ассоциирование специально созданной структуры с функцией решает многие проблемы использования обобщенных процедур и контейнеров, поэтому применение обобщенного компонента – подходящий случай воспользоваться на практике всего приемами работы со структурами. Структуры – это еще и удобное место для размещения кода ошибки, который не придется втискивать в аргументы функции. Так что небольшая трата времени на написание определения типа окупается с лихвой.

Глава 11

Объектно-ориентированное программирование на С

Мы ценим простое выражение сложных идей.
...
Мы за плоские формы,
Потому что они разрушают иллюзии и показывают голую правду.

— Le Tigre «Slideshow at Free University»

Вот как устроена типичная библиотека на С или любом другом языке:

- небольшой набор структур данных, представляющих ключевые понятия предметной области, к которой относится библиотека;
- набор функций (их часто называют *интерфейсными*) для манипуляции этими структурами данных.

Например, в библиотеке для работы с XML будут структуры для описания XML-документа и, возможно, его представлений, а также многочисленные функции для перехода между структурой данных в памяти и XML-файлом на диске, для поиска элементов в структуре и т. д. В библиотеке для работы с базами данных будут структуры для представления состояния взаимодействия с базой и, возможно, для представления таблиц, а также функции для отправки запросов к базе и разбора полученных от нее данных.

Это вполне разумный способ организации программы или библиотеки. Он позволяет автору выражать концепции с помощью глаголов и существительных, характерных для решаемой задачи.

Я не буду тратить времени (и разжигать страсти) на формулирование точного определения объектно-ориентированного программирования (ООП), но из приведенного выше описания объектно-ориентированной библиотеки должно быть ясно, какую цель мы преследуем: несколько центральных структур данных, с каждой из которых связан набор функций для выполнения операций над этой структурой.

Для любого эксперта, считающего, что некоторая возможность – неотъемлемая характеристика ООП, найдется другой эксперт, который полагает, что она толь-

ко отвлекает от существа ООП[1]. Тем не менее перечислю несколько дополнений к основной идее о структуре и функциях, которые распространены очень широко.

- Наследование, позволяющее расширить структуру, добавив к ней новые элементы.
- Виртуальные функции, которые определяют поведение любого объекта класса, но могут быть переопределены для других экземпляров данного класса или его потомков в дереве наследования.
- Точное управление областью видимости, в частности разделение элементов структуры на открытые и закрытые.
- Перегрузка операторов – синтаксический прием, позволяющий изменять семантику операции в зависимости от типов операндов.
- Механизм подсчета ссылок, позволяющий освобождать объект тогда и только тогда, когда ни один из связанных с ним ресурсов более не используется.

В этом разделе мы рассмотрим, как эти идеи можно реализовать на С. Ни одна из них не представляет особых трудностей: для подсчета ссылок нужно хранить счетчик, для перегрузки функций (но не операторов) используется ключевое слово `_Generic`, специально предназначенное для этой цели, а виртуальные функции можно реализовать с помощью функции диспетчеризации, возможно, дополненной индексированной таблицей альтернативных функций.

И это подводит нас к интересному вопросу: если все эти расширения базовой идеи о структуре и связанных с ней функциях так просто реализовать и для этого требуется всего несколько строк кода, то почему пишущие на С так редко ими пользуются?

Гипотеза Сепира-Уорфа о связи языка с мышлением и способом познания реальности, формулировалась по-разному; мне нравится формулировка, согласно которой одни языки заставляют нас размышлять о таких предметах, которые в других языках даже не рассматриваются. Многие языки вынуждают принимать во внимание пол, потому что на них очень трудно построить высказывание о человеке, не употребляя признаков пола: *он*, *она*, *его* или *ее*. С заставляет думать о выделении памяти больше, чем другие языки (и это порождает у тех, кто на С не пишет, ложные стереотипы, будто весь код на С – не что иное, как манипулирование памятью). В языках, где имеются развитые средства областей видимости, приходится внимательно следить за тем, когда и каким объектам видна переменная, – даже если технически язык позволяет откомпилировать программу, в которой все члены объекта объявлены открытыми, кто-нибудь наверняка обзовет вас лентяем и напомнит о нормах языка, требующих выбирать как можно более узкую область видимости.

Поэтому программирование на С ставит нас в привилегированное положение, чего не случилось бы, если бы мы писали на каком-нибудь официальном объект-

---

[1] Однажды я присутствовал на собрании группы пользователей Java, где с основным докладом выступал Джеймс Гослинг (автор Java). Когда начали задавать вопросы, кто-то спросил: «Если бы вы могли создать Java еще раз, что бы вы изменили?» Гослинг ответил: «Я бы выкинул классы». Allen Holub «Why extends is evil».

но-ориентированном языке типа C++ или Java: мы можем реализовать целый ряд расширений базового объекта, состоящего из структуры и функций, пользуясь простыми средствами, но не обязаны это делать и можем отказаться от этого решения, если оно только усложняет программу, не принося ощутимой выгоды.

Расширение структур и словарей

В начале этого раздела я приведу пример типичной схемы организации библиотеки: структура плюс набор функций для операций с ней. Но, как следует из названия, раздел все же посвящен расширениям: как добавлять новые элементы в структуру и как добавлять новые функции, которые будут правильно работать с уже существующими и с новыми структурами?

В 1936 году, исследуя формальную математическую задачу (проблему разрешения), Алонсо Чёрч разработал *лямбда-исчисление* – формальный аппарат описания функций и переменных. В 1937 году, работая над той же задачей, Алан Тьюринг описал формальный язык в виде машины, оснащенной лентой и головкой, которая могла двигаться вдоль ленты для чтения и записи данных. Впоследствии была доказана эквивалентность лямбда-исчисления Чёрча и машины Тьюринга – любое вычисление, которое можно выразить с помощью одного из этих средств, можно выразить и с помощью другого. С тех пор конструкции, придуманные Чёрчем и Тьюрингом, лежат в основе различных способов структурирования данных.

Лямбда-исчисление опирается на списки именованных элементов; в псевдокоде, написанном на его основе, сведения о человеке можно было бы представить так:

```
(person (
  (name "Sinead")
  (age 28)
  (height 173)
))
```

В случае машины Тьюринга мы резервируем для структуры место на ленте. В первых нескольких позициях будет записано имя, затем – возраст и т. д. Спустя почти сто лет лента машины Тьюринга по-прежнему остается неплохим описанием памяти компьютера: вспомните раздел «Все, что нужно знать об арифметике указателей» выше, где говорилось, что база плюс смещение – это и есть основа построения структур на C. Когда мы пишем:

```
typedef struct {
  char * name;
  double age, height;
} person;

person sinead = {.name="Sinead", .age=28, .height=173};
```

`sinead` указывает на начало блока памяти, а `sinead.height` – на позицию ленты, следующую сразу после `name` и `age` (с учетом дополнительных позиций для выравнивания).

Остановлюсь на некоторых различиях между представлением в виде списка и в виде блока памяти.

- Смещение на определенное расстояние от некоторого адреса и по сей день остается одной из самых быстрых операций компьютера. Компилятор C даже транслирует имена полей в смещения. Напротив, поиск чего-то в списке требует просмотра: если дано имя поля "age", то какой элемент списка ему соответствует и где он находится в памяти? В каждой системе есть свои способы сделать такой поиск максимально быстрым, но все равно он неизбежно занимает больше времени, чем простое смещение от базы.
- Добавить новый элемент в список гораздо проще, чем новое поле в структуру, состав которой фиксирован на этапе компиляции.
- Компилятор C может сообщить, что `hieght` – опечатка; для этого ему нужно лишь заглянуть в определение структуры и убедиться, что такого элемента в нем нет. Но список допускает расширение, поэтому мы не можем узнать, есть в нем элемент `hieght` или нет, пока не запустим программу и не проверим.

Последние два пункта демонстрируют некое противоречие: с одной стороны, нам нужна расширяемость, которая позволила бы добавлять поля в структуру, а с другой – регистрация, позволяющая сообщать об ошибке, когда производится обращение к полю, отсутствующему в структуре. Приходится искать компромисс, и существуют различные подходы к контролируемому расширению существующего списка.

В C++, Java и производных от них языках имеется синтаксис для порождения нового типа, расширяющего существующий. При этом сохраняются скорость, обусловленная простым смещением от базы, и проверка на этапе компиляции, но ценой повышенного многословия: если в C имеется лишь `struct` с простейшими правилами видимости (см. раздел «Область видимости» ниже), то в Java мы видим ключевые слова `implements`, `extends`, `final`, `instanceof`, `class`, `this`, `interface`, `private`, `public`, `protected`.

Perl, Python и многие языки, берущие начало от LISP, основаны на списках именованных элементов, и именно так реализуют структуру. Для расширения списка достаточно добавить в него новые элементы. Плюсы: полная расширяемость за счет добавления именованного элемента. Минусы: отсутствие регистрации и, несмотря на различные ухищрения, значительное снижение скорости, по сравнению со сложением смещения и базы. Во многих языках из этого семейства есть система определения классов, позволяющая зарегистрировать некий набор элементов списка и затем проверять, соответствует ли использование определению. При правильном применении это обеспечивает приемлемый компромисс между простотой расширения и проверкой корректности. Но вернемся к старому доброму C; его структуры дают самый быстрый способ доступа к полям, и мы получаем проверку на этапе компиляции ценой утраты расширяемости на этапе выполнения. Если требуется гибкий список, который может расти во время выполнения, то необходима какая-то структура данных вроде хэшей GLib или описываемого ниже простого словаря.

Реализация словаря

Построить словарь с теми средствами, которые имеются в основанном на структурах языке С нетрудно. И это даст нам хорошую возможность создать объект и продемонстрировать, как на практике работает концепция «структура плюс функции», которой и посвящена эта глава. Отметим, однако, что другие авторы уже претворили рассмотренную идею словаря в жизнь и тщательно протестировали ее; см., например, индексированные таблицы данных GHashTable в библиотеке GLib. Мы лишь хотим показать, что наличия составных структур и простых массивов достаточно для реализации объекта «словарь».

Мы начнем с простой пары ключ/значение. Необходимые для работы с ней функции находятся в файле *keyval.c*. В примере 11.1 приведен заголовок с объявлениями всех интерфейсных функций.

Пример 11.1 ❖ Заголовок – открытый интерфейс класса ключей и значений (keyval.h)

```
typedef struct keyval{
  char *key;
  void *value;
} keyval;

keyval *keyval_new(char *key, void *value);
keyval *keyval_copy(keyval const *in);
void keyval_free(keyval *in);
int keyval_matches(keyval const *in, char const *key);
```

Читатели с опытом работы на традиционных объектно-ориентированных языках программирования увидят здесь много знакомых черт. При разработке нового объекта первое побуждение – написать функции создания, копирования и освобождения, что в этом примере и сделано. Затем обычно идут функции, реализующие специфику работы со структурой, в данном случае функция `keyval_matches`, которая проверяет, совпадает ли ключ в паре `keyval` с переданной строкой.

Наличие функций new/copy/free означает, что вам не придется мучиться с управлением памятью: в функциях new и copy память выделяется с помощью `malloc`, а в функции free – освобождается с помощью `free`. Отныне код, в котором используется объект, никогда не будет вызывать для него `malloc` и `free` явно, доверяя управление памятью функциям `keyval_new`, `keyval_copy` и `keyval_free`.

Пример 11.2 ❖ Типичный шаблон кода объекта: структура плюс функции создания, копирования и освобождения (keyval.c)

```
#include <stdlib.h> //malloc
#include <strings.h> //strcasecmp (описана в POSIX)
#include "keyval.h"

keyval *keyval_new(char *key, void *value){
  keyval *out = malloc(sizeof(keyval));
  *out = (keyval){.key = key, .value=value};      ❶
  return out;
}
```

```
/** Копировать пару ключ/значение. В новой паре хранятся указатели на
    данные из старой пары, а не копии данных. */
keyval *keyval_copy(keyval const *in){
  keyval *out = malloc(sizeof(keyval));
  *out = *in;                                  ❷
  return out;
}

void keyval_free(keyval *in){ free(in); }

int keyval_matches(keyval const *in, char const *key){
  return !strcasecmp(in->key, key);
}
```

❶ Позиционные инициализаторы облегчают заполнение структуры.

❷ Напомним, что для копирования содержимого структуры достаточно знака равенства. Если бы мы хотели скопировать также данные, на которые ведут указатели, хранящиеся в структуре (а не просто сами указатели), то после этой строки пришлось бы написать дополнительный код.

Теперь у нас есть объект, представляющий одну пару ключ/значение, и мы можем перейти к организации словаря как списка таких пар. В примере 11.3 приведен заголовок.

Пример 11.3 ❖ Открытый интерфейс словаря (dict.h)

```
#include "keyval.h"

extern void *dictionary_not_found; ❶

typedef struct dictionary{
  keyval **pairs;
  int length;
} dictionary;

dictionary *dictionary_new (void);
dictionary *dictionary_copy(dictionary *in);
void dictionary_free(dictionary *in);
void dictionary_add(dictionary *in, char *key, void *value);
void *dictionary_find(dictionary const *in, char const *key);
```

❶ Это признак, возвращаемый, когда ключ не найден в словаре. Он должен быть открытым.

Как видите, здесь присутствуют все те же функции new/copy/free, а также функции, реализующие специфику словаря. И еще некий признак, о котором мы поговорим ниже. В примере 11.4 показана закрытая реализация.

Пример 11.4 ❖ Реализация словаря (dict.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "dict.h"
```

```
void *dictionary_not_found;

dictionary *dictionary_new (void){
  static int dnf;                                                    ❶
  if (!dictionary_not_found) dictionary_not_found = &dnf;
  dictionary *out= malloc(sizeof(dictionary));
  *out= (dictionary){ };
  return out;
}

static void dictionary_add_keyval(dictionary *in, keyval *kv){       ❷
  in->length++;
  in->pairs = realloc(in->pairs, sizeof(keyval*)*in->length);
  in->pairs[in->length-1] = kv;
}

void dictionary_add(dictionary *in, char *key, void *value){
  if (!key){fprintf(stderr, "Ключ не может быть равен NULL.\n"); abort();} ❸
  dictionary_add_keyval(in, keyval_new(key, value));
}

void *dictionary_find(dictionary const *in, char const *key){
  for (int i=0; i< in->length; i++)
    if (keyval_matches(in->pairs[i], key))
        return in->pairs[i]->value;
  return dictionary_not_found;
}

dictionary *dictionary_copy(dictionary *in){
  dictionary *out = dictionary_new();
  for (int i=0; i< in->length; i++)
    dictionary_add_keyval(out, keyval_copy(in->pairs[i]));
  return out;
}

void dictionary_free(dictionary *in){
  for (int i=0; i< in->length; i++)
    keyval_free(in->pairs[i]);
  free(in);
}
```

❶ Вполне может оказаться, что значением ключа является `NULL`, поэтому необходим уникальный признак, обозначающий отсутствие ключа. Я не знаю, где в памяти хранится переменная `dnf`, но ее адрес заведомо уникален.

❷ Напомню, что функция, помеченная ключевым словом `static`, не видна вне файла, так что это еще одно напоминание о том, что она является закрытой частью реализации.

❸ Признание в грехе: использование `abort` – дурной стиль; лучше было бы написать какой-нибудь макрос типа того, что приведен в файле `stopif.h`. Я поступил так, чтобы продемонстрировать одну особенность тестовой оснастки.

Теперь у нас есть словарь, и в примере 11.5 показано, как пользоваться им, не заботясь об управлении памятью, возложенном на функции new/copy/free/add, и не

работая явно с парами ключ/значение, поскольку они находятся на уровне ниже того, который нас интересует.

Пример 11.5 ❖ Пример использования словарного объекта; копаться в потрохах структуры нет нужды, так как интерфейсные функции предоставляют все необходимое (dict_use.c)

```
#include <stdio.h>
#include "dict.h"

int main(){
  int zero = 0;
  float one = 1.0;
  char two[] = "two";
  dictionary *d = dictionary_new();
  dictionary_add(d, "an int", &zero);
  dictionary_add(d, "a float", &one);
  dictionary_add(d, "a string", &two);
  printf("Я сохранил целое:%i\n",
        *(int*)dictionary_find(d, "an int"));
  printf("Сохраненная строка:%s\n", (char*)dictionary_find(d, "a string"));
  dictionary_free(d);
}
```

Таким образом, написания структуры с функциями new/copy/free и прочими оказалось достаточно для обеспечения нужного уровня инкапсуляции: словарю безразлично внутреннее устройство пары ключ/значение, а приложение может не думать об устройстве словаря.

Стереотипный код не так плох, как в некоторых других языках, но, конечно, имеется некое повторение при реализации функций new/copy/free. По мере развития примера мы еще не раз столкнемся с этим шаблоном.

Когда мне наскучило повторять один и тот же код, я даже написал несколько макросов для его автоматической генерации. Например, функции копирования различаются только способом работы с внутренними указателями, поэтому можно написать макрос, который автоматизирует создание всего стереотипного кода, не относящегося к внутренним указателям:

```
#define def_object_copy(tname, ...)       \
  void * tname##_copy(tname *in) {        \
    tname *out = malloc(sizeof(tname));   \
    *out = *in;                           \
    __VA_ARGS__;                          \
    return out;                           \
  }

def_object_copy(keyval) // После расширения получается показанная выше функция keyval_copy.
```

Но устранение избыточности – не то, о чем следует печься в первую очередь. И хотя эстетическое чувство математика взывает к минимизации повторений и уменьшению количества текста на странице, иногда «лишний» код делает программу более понятной и надежной.

С без зазоров

В языке С единственный способ добавить новые элементы в структуру – обернуть ее другой структурой. Допустим, что имеется некий тип:

```
typedef struct {
  ...
} list_element_s;
```

который уже включен в производственную систему и изменению не подлежит. А нам все же необходимо добавить новое поле – маркер типа. Тогда придется определить новую структуру:

```
typedef struct {
  list_element_s elmt;
  char typemarker;
} list_element_w_type_s;
```

Плюсы: способ тривиальный, а скорость доступа сохраняется. Минусы: для обращения к имени элемента понадобится каждый раз выписывать полный путь `your_typed_list->elmt->name`, тогда как при расширении класса в C++ и Java было бы достаточно написать `your_typed_list->name`. Добавьте еще несколько слоев обертки, и это станет надоедать. В разделе «Указатели без malloc» на стр. 142 мы видели, как псевдонимы позволяют бороться с этой напастью.

В C11 использование вложенных структур упрощено благодаря разрешению включать в структуру анонимные элементы. И хотя в стандарт эта возможность была добавлена только в декабре 2011 года, в компиляторе Microsoft она была реализована гораздо раньше, а в gcc для ее активации нужно задать флаг `--ms-extensions` в командной строке.

Синтаксис: включите какой-либо структурный тип в любое место объявления новой структуры, как мы поступили со структурой `point` в примере 11.6, но не задавайте имя этого элемента[1]. В примере 11.6 указано только имя структурного типа `struct point`, тогда как именованное объявление имело бы вид `struct point elementname`. Все элементы вложенной структуры включаются в новую структуру, как если бы они были объявлены непосредственно в обертывающей структуре.

В примере 11.6 структура, описывающая точку на плоскости, расширяется до описания точки в трехмерном пространстве. Этот пример примечателен только тем, насколько естественно `threepoint` расширяет `point`, – до такой степени, что пользователи структуры `threepoint` даже не заметят, что ее определение основано на другой структуре.

---

[1] Приведем цитату из стандарта C11 §6.7.2.1(13): «Безымянный член [struct или union], для которого в качестве спецификатора типа указан структурный спецификатор без тега, называется анонимной структурой... Члены анонимной структуры или объединения (union) считаются членами объемлющей структуры или объединения. Это правило применяется рекурсивно, если объемлющая структура или объединение сама является анонимной».

Пример 11.6 ❖ Анонимная подструктура внутри обертывающей структуры естественно встраивается в обертку (seamlessone.c)

```
#include <stdio.h>
#include <math.h>

typedef struct point {
  double x, y;
} point;

typedef struct {
  struct point;                                   ❶
  double z;
} threepoint;

double threelength (threepoint p){
  return sqrt(p.x*p.x + p.y*p.y + p.z*p.z);       ❷
}

int main(){
  threepoint p = {.x=3, .y=0, .z=4};              ❸
  printf("p отстоит на%g единиц длины от начала координат\n",
         threelength(p));
}
```

❶ Это анонимная подструктура. В неанонимной версии она имела бы имя, например struct point twopt.

❷ Поля x и y структуры point выглядят и ведут себя в точности так же, как дополнительное поле z структуры threepoint.

❸ Даже в объявлении нет и намека на то, что x и y унаследованы от существующей структуры.



В стандарте запрещается использовать typedef во вложенном анонимном объявлении. Комитет по стандартизации явно отклонил эту возможность, создав тем самым одно из немногих мест в языке C, где псевдоним структурного типа не может заменить самого определения структуры[1]. Впрочем, если вы применяете соглашение об именовании, описанное в разделе «Псевдонимы типов спешат на помощь», то это просто означает, что нужно будет добавить слово struct перед именем структурного типа.

А теперь о приеме, благодаря которому этот механизм становится действительно полезным. В исходном объекте point, скорее всего, имелся набор интерфейсных функций, потенциально все еще полезных, например функция length, возвращающая длину от точки до начала координат. Как воспользоваться этой функцией, если у части большей структуры нет имени?

Решение простое: взять анонимное объединение структур с именованной и неименованной частями point. Поскольку структуры идентичны, то все их поля занимают в точности одинаковые позиции в объединении и отличаются только

---

[1] Обсуждение взято из статьи Дэвида Кэмерона «Clarifications to Anonymous Structures and Unions", WG14/N1549, 22 декабря 2010 года. Голосование и одобрение состоялись на заседании комитета в марте 2011 года.

именами. Когда требуется вызвать функцию, работающую с исходной структурой, мы будем пользоваться именованной частью, а для органичного встраивания подструктуры в объемлющую структуру – анонимной. В примере 11.7 код из примера 11.6 переписан с использованием этой идеи.

Пример 11.7 ❖ **Структура** `point` **органично встраивается в** `threepoint`**, и у нас по-прежнему есть имена, благодаря которым можно использовать функции, рассчитанные на** `point` **(seamlesstwo.c)**

```
#include <stdio.h>
#include <math.h>

typedef struct point {
  double x, y;
} point;

typedef struct {
  union {
    struct point;                                    ❶
    point p2;                                        ❷
  };
  double z;
} threepoint;

double length (point p){
  return sqrt(p.x*p.x + p.y*p.y);
}

double threelength (threepoint p){
  return sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
}

int main(){
  threepoint p = {.x=3, .y=0, .z=4};                 ❸
  printf("p отстоит на%g единиц длины от начала координат\n",
         threelength(p));
  double xylength = length(p.p2);                    ❹
  printf("Ее проекция на плоскость XY отстоит на%g единиц длины "
         "от начала координат\n", xylength);
}
```

❶ Это анонимная структура.

❷ Это именованная структура. Будучи частью объединения, она идентична анонимной структуре, отличаясь только наличием имени.

❸ Структура `point` по-прежнему органично включается в структуру `threepoint`, но...

❹ ... элемент `p2` является именованным, каким всегда и был, поэтому мы можем использовать его для вызова интерфейсных функций, написанных для работы с исходной структурой.

После объявления `threepoint p` мы можем ссылаться на координату *X* как по имени `p.x` (из-за наличия анонимной структуры), так и по имени `p.p2.x` (из-за

наличия именованной структуры). В последней строке примера расстояние от проекции точки на плоскость *XY* до начала координат вычисляется с помощью `length(p.p2)`.

Начиная с этого момента, все возможности распространяются и на пространство $\mathbb{R}^3$. Мы можем таким образом расширить любую структуру и продолжать пользоваться функциями, написанными для исходного объекта.



Вы обратили внимание, что я впервые в этой книге употребил ключевое слово `union`? Объединения – это одна из тех вещей, объяснить которые очень просто: это почти то же, что структура, только все элементы занимают одно и то же место в памяти, – но о подводных камнях можно написать несколько страниц. Память нынче дешева, и при разработке приложений нам нет нужды заботиться о выравнивании, поэтому лучше ограничиться структурами – это уменьшит шансы допустить ошибку, пусть даже в каждый момент времени используется только один элемент.

Наследование нескольких структур таким способом – вещь рискованная. Выберите какую-нибудь одну структуру, которая станет базой расширения, воспользовавшись трюком с объединением, а остальные расширяйте с помощью обычных подструктур. Например, в библиотеке GNU Scientific Library есть типы матрицы и вектора (причем для `struct _gsl_matrix` имеется псевдоним `gsl_matrix`). Предположим, что их надо поместить в одну структуру.

```
typedef struct{        //Увы, так не получится.
struct _gsl_vector;
struct _gsl_matrix;
} data_set;

data_set d;
```

Выглядит совершенно невинно, но лишь до тех пор, пока вы не обнаружите, что в обеих структурах `_gsl_vector` и `_gsl_matrix` имеется поле `data`. На какой элемент `data` ссылается `d.data`: из матрицы или из вектора? Не существует синтаксиса, позволяющего избирательно включать или переименовывать элементы структуры, поэтому лучшее, что можно сделать, – взять что-то одно (матрицу или вектор) за основу, а вторую структуру сделать вспомогательной, допускающей обращение к подэлементу только по имени:

```
typedef struct{               // Вектор со вспомогательной матрицей
struct _gsl_vector;           // Анонимная, органично встроенная
struct _gsl_matrix matrix;    // Именованная
} data_set;
```

Стройте код на основе указателей на объекты

В главе 10 были в основном представлены технические приемы, касающиеся структур данных, а не указателей на них, тогда как лейтмотивом этой главы является объявление и использование указателей на структуры.

На самом деле если вы работаете с обычной структурой, то функции new/copy/free получаются автоматически:

new

Пользуйтесь позиционными инициализаторами в первой же строке той части кода, где необходима структура. Дополнительный бонус: структуры можно объявлять на

этапе компиляции, так что они становятся доступны сразу же, без обращения к функции инициализации.

copy
: Для копирования достаточно знака равенства.

free
: Не о чем беспокоиться; скоро структура покинет область видимости.

Таким образом, работая с указателями, мы сами усложняем себе жизнь. И тем не менее есть общее мнение, что именно указатели следует класть в основу дизайна программ. Перечислим их преимущества.

- Копирование указателя обходится дешевле копирования структуры целиком, поэтому мы экономим микросекунду на каждом обращении к функции, принимающей структуру. Разумеется, это становится ощутимо только после нескольких миллиардов обращений.
- Все библиотеки для работы со структурами данных (к примеру, деревьями и связанными списками) написаны в расчете на указатели.
- Когда мы заполняем дерево или список, автоматическое уничтожение структуры по выходе из области видимости, в которой она была создана, отнюдь не всегда желательно.
- Многие функции, принимающие структуру, модифицируют ее содержимое, а это означает, что без передачи указателя все равно не обойтись. Когда одни функции принимают структуру, а другие – указатель на структуру, возникает путаница (я сам писал такие интерфейсы и сожалею об этом), поэтому лучше уж всегда передавать указатель.
- Если какое-то поле структуры содержит указатель на данные, то все преимущества использования обычных структур улетучиваются: для выполнения глубокого копирования (когда копируются не только указатели, но и данные, на которые они указывают) потребуется функция копирования и, скорее всего, также функция освобождения, гарантирующая уничтожение внутренних данных.
- Не существует правил использования структур, пригодных на все случаи жизни. Когда в процессе эволюции одноразовая структура, созданная для упрощения внутренних операций, становится основой организации данных, достоинства указателей выходят на первый план, а достоинства структур как таковых отступают.

Функции в структурах

До сих пор мы во всех заголовках видели структуру и за ней набор функций, но ведь структура может включать указатели на функции в качестве членов, поэтому все функции, кроме `object_new`, можно было бы перенести в саму структуру:

```
typedef struct keyval{
  char *key;
  void *value;
  keyval *(*keyval_copy)(keyval const *in);
  void (*keyval_free)(keyval *in);
  int (*keyval_matches)(keyval const *in, char const *key);
} keyval;

keyval *keyval_new(char *key, void *value);
```

Отбрасывание указателя

Допустим, имеется указатель на функцию `fn`, то есть `*fn` – это функция, а `fn` – ее адрес в памяти. Тогда запись `(*fn)(x)` осмысленна и означает вызов функции, но что могла бы значить запись `fn(x)`? В данном случае С следует подходу «делай, что я имел в виду» и

интерпретирует вызов указателя на функцию как обычный вызов функции. Для этого даже есть специальный термин – *отбрасывание указателя* (pointer decay). Именно поэтому в этой книге я трактую функции и указатели на функции одинаково.

По большей части, это вопрос стиля, влияющий лишь на то, как мы ищем функции в документации и как выглядит код на странице. Кстати, с точки зрения документации я предпочитаю схему именования `keyval_copy` схеме `copy_keyval`: в первом случае в алфавитном указателе все функции, ассоциированные с `keyval`, будут собраны в одном месте.

Реальное преимущество включения функций в саму структуру заключается в том, что таким образом становится проще изменить функцию, ассоциированную с каждым экземпляром. В примере 11.8 показана простая структура списка, настолько общая, что в ней можно хранить рекламные объявления, тексты песен, кулинарные рецепты и вообще все, что угодно. Кажется естественным печатать столь различные типы списков в разном формате, поэтому мы должны иметь возможность определить несколько функций печати.

Пример 11.8 ❖ Довольно общая структура со встроенным методом печати (print_typedef.c)

```
#ifndef textlist_s_h
#define textlist_s_h
typedef struct textlist_s {
  char *title;
  char **items;
  int len;
  void (*print)(struct textlist_s*);
} textlist_s;
#endif
```

В примере 11.9 с помощью этого псевдонима типа объявляются и используются два объекта. При создании объектов задаются различные методы печати.

Пример 11.9 ❖ Включение функции в состав структуры ясно показывает, какая функция с какой структурой используется (print_methods.c)

```
#include <stdio.h>
#include "print_typedef.h"

static void print_ad(textlist_s *in){
  printf("BUY THIS%s!!!! Features:\n", in->title);
  for (int i=0; i< in->len; i++)
    printf("·%s\n", in->items[i]);
}

static void print_song(textlist_s *in){
  printf("♫%s ♫\nLyrics:\n\n", in->title);
  for (int i=0; i< in->len; i++)
    printf("\t%s\n", in->items[i]);
}

textlist_s save = {.title="God Save the Queen",
```

```
    .len=3, .items=(char*[]){
    "There's no future", "No future", "No future for me."},
    .print=print_song};         ❶

textlist_s spend = {.title="Never mind the Bollocks LP",
    .items=(char*[]){"By the Sex Pistols", "Anti-consumption themes"},
    .len=2, .print=print_ad};

#ifndef skip_main
int main(){
  save.print(&save);            ❷
  printf("\n-----\n\n");
  spend.print(&spend);
}
#endif
```

❶ Обратите внимание: именно здесь функция включается в структуру, а несколькими строчками ниже то же самое делается для функции spend.

❷ Вызовы всех методов, включенных в структуру, выглядят одинаково. Нам не нужно помнить, что save – это текст песни, а spend – рекламное объявление.

Последние три строчки открывают путь к определению единообразного интерфейса к совершенно различным функциям. Любая функция, принимающая указатель textlist_s* в аргументе t, может выполнить такой вызов: t->print(&t).

Правда, есть и недостаток: мы рискуем нарушить правило, согласно которому вещи, предназначенные для разных целей, должны и выглядеть по-разному. Если какая-нибудь функция, сохраненная в поле print, имеет тонкие побочные эффекты, то никакого предупреждения вы не получите.

Обратите внимание на ключевое слово static, которое означает, что никакой код вне этого файла не сможет вызвать функцию print_song или print_ad по имени. Однако ту и другую можно вызвать косвенно: save.print или spend.print соответственно.

Хотелось бы добавить кое-какие «бантики». Во-первых, в записи save.print(&save) два раза повторяется слово save. Было бы прекрасно, если бы мы могли написать просто save.print(), а система догадалась бы, что первым аргументом должен быть объект, произведший обращение. Функция могла бы распознавать специальную переменную с именем this или self или можно было бы ввести в язык соглашение: компилятор переписывает object.fn(x) в виде fn(object, x).

Увы, но это уже будет не язык C.

В C нет никаких волшебных переменных, он честно говорит правду о том, какие параметры передаются функции. Обычно для манипуляций с параметрами функции мы обращаемся к услугам препроцессора, который с большим удовольствием заменит f(anything) на f(anything else). Однако все преобразования применяются к тому, что уже есть внутри скобок. Никаким способом нельзя заставить препроцессор преобразовать s.prob(d) в s.prob(s, d). Если вы готовы отказаться от рабского копирования синтаксиса C++, то можете написать такие макросы:

```
#define Print(in) (in).print(&in)
#define Copy(in, ...) (in).copy((in), __VA_ARGS__)
#define Free(in, ...) (in).free((in), __VA_ARGS__)
```

Но теперь глобальное пространство имен засорено символами Print, Copy и Free. Быть может, в вашем случае это и оправдано (особенно если учесть, что с любым объектом должны быть ассоциированы функции копирования и освобождения).

Организация пространства имен должна быть продумана, так чтобы не было конфликтов имен, поэтому и макросы следует называть соответственно:

```
#define Typelist_print(in) (in).estimate(&in)
#define Typelist_copy(in, ...) (in).copy((in), __VA_ARGS__)
```

Но вернемся к типу typelist_s. У нас есть способ распечатать тексты песен и рекламные объявления. Ну а как насчет кулинарных рецептов или еще чего-нибудь? И что случится, если программист включит в структуру список, но забудет добавить нужную функцию?

Необходим какой-то метод по умолчанию, и реализовать это легко с помощью функции диспетчеризации. Эта функция проверит наличие во входной структуре метода print, и если соответствующее поле отлично от NULL, то вызовет найденный метод. В противном случае она предоставит метод по умолчанию. В примере 11.10 продемонстрирована такая функция диспетчеризации, которая правильно распечатывает объект песни с помощью включенного в него метода печати, но для рецепта, у которого метода печати нет, сама реализует простую печать по умолчанию.

Пример 11.10 ❖ **У объекта recipe нет метода print, но функция диспетчеризации все равно распечатывает его (print_dispatch.c)**

```
#define skip_main
#include "print_methods.c"

textlist_s recipe = {.title="Starfish and Coffee",
    .len=2, .items=(char*[]){"Starfish", "Coffee"}};

void textlist_print(textlist_s *in){
  if (in->print){
    in->print(in);
    return;
  }

  printf("Название:%s\n\nЭлементы:\n", in->title);
  for (int i=0; i< in->len; i++)
    printf("\t%s\n", in->items[i]);
}

int main(){
  textlist_print(&save);
  printf("\n-----\n\n");
  textlist_print(&recipe);
}
```

Таким образом, функции диспетчеризации предлагают процедуру по умолчанию, разрешают проблему с отсутствием ключевого слова `this` или `self` и при этом устроены по аналогии с обычными интерфейсными функциями типа `textlist_copy` или `textlist_free` (если бы таковые были определены).

Есть и другие способы достичь той же цели. Выше я использовал позиционные инициализаторы для заполнения структуры, поэтому незаданные элементы оказывались равными `NULL`, и применение функции диспетчеризации имело прямой смысл. Если бы мы потребовали от пользователя создавать объект только с помощью функции `textlist_new`, то можно было бы установить в ней функции по умолчанию. Тогда исключить избыточность из конструкции `save.print(&save)` можно было бы с помощью простого макроса, как и раньше.

Повторю – всегда есть разные варианты. У нас уже больше чем достаточно синтаксических средств для единообразного вызова различных функций при работе с различными объектами. Осталась трудная часть: написать эти различные функции, так чтобы при единообразном вызове они вели себя, как ожидает пользователь.

V-таблицы

Предположим, что с момента первоначального проектирования структуры `textlist_s` прошло некоторое время и возникли новые потребности. Хотелось бы иметь возможность публиковать списки во Всемирной паутине, но для этого их нужно представить в формате HTML. Как добавить новые функции печати в существующую структуру?

Мы уже видели (раздел «С без зазоров»), как можно расширять структуры, и эту идею можно использовать для организации вложенной структуры, содержащей новые функции.

В этом разделе описывается альтернатива: добавлять новые функции вне структуры объекта. Они хранятся в так называемой *виртуальной таблице* (или *v-таблице*); название напоминает нам о виртуальных функциях из объектно-ориентированного лексикона, а также отдает дань памяти сложившейся в 1990-х годах моде называть *виртуальным* все, что реализовано программно. V-таблица представляет собой простую хэш-таблицу ключей и значений. В разделе «Реализация словаря» выше было показано, как построить такую таблицу, но сейчас я воспользуюсь хэшами из библиотеки Glib.

По данному объекту я сгенерирую хэш (ключ) и ассоциирую с ним функцию. Затем, когда пользователь вызовет функцию диспетчеризации для некоторой операции, эта функция поищет нужную операцию в хэш-таблице; если найдет, то вызовет ее, иначе выполнит действия по умолчанию.

Вот что нам понадобится для реализации этого плана.

- Функция хэширования.
- Средство проверки типов. Необходимо гарантировать, что сигнатуры всех функций, хранящихся в хэш-таблице, одинаковы.
- Таблица ключей и значений и ассоциированные с ней функции добавления и поиска.

Функция хэширования

Функция хэширования сопоставляет входному параметру число таким образом, что вероятность получить одно и то же число для разных параметров близка к нулю.

В библиотеке Glib есть несколько готовых функций хэширования, в том числе `g_direct_hash`, `g_int_hash` и `g_str_hash`. Первая (непосредственное хэширование) предназначена для указателей, она просто трактует указатель как результирующее число; в этом случае конфликт возможен, только если оба объекта занимают одно и то же место в памяти.

В более сложных ситуациях можно написать функции хэширования самостоятельно. Ниже приведена широко распространенная функция, авторство которой приписывается Дэниэлу Дж. Бернстайну. Для каждого символа в строке (или каждого байта многобайтного символа UTF-8) вычисленный ранее результат умножается на 33 и к произведению прибавляется текущий символ (или байт). С высокой вероятностью произойдет переполнение переменной типа `unsigned int`, в которой хэш сохраняется, но это всего лишь еще один неявный детерминированный шаг алгоритма.

```
static unsigned int string_hash(char const *str){
    unsigned int hash = 5381;
    char c;
    while ((c = *str++)) hash = hash*33 + c;
    return hash;
}
```

Впрочем, Glib уже предоставляет функцию `g_str_hash`, так что использовать приведенную выше функцию необязательно. Однако ее можно взять за образец для написания альтернативных функций хэширования. Так, список указателей можно хэшировать следующим образом:

```
static unsigned int ptr_list_hash(void const **in){
    unsigned int hash = 5381;
    void *c;
    while ((c = *in++)) hash = hash*33 + (uintptr_t)c;
        return hash;
}
```

Для читателей с подготовкой в объектно-ориентированном программировании отметим, что мы уже прошли большую часть пути, ведущего к реализации множественной диспетчеризации. Дайте мне два объекта, и я смогу вычислить совместный хэш-код указателей на них и ассоциировать с ним функцию в таблице ключей и значений.

Для хэш-таблиц Glib необходимо сравнение на равенство, поэтому библиотека включает функции сравнения `g_direct_equal`, `g_int_equal` и `g_str_equal`, соответствующие функциям хэширования.

При любом выборе функции хэширования вероятность коллизий остается, хотя, если функция написана правильно, она близка к нулю. Я использую в своем коде функции хэширования, аналогичные вышеупомянутым, и сознаю, что в один пре-

красный день кто-нибудь особо неудачливый может наткнуться на два набора указателей, приводящих к коллизии хэша. Но решая, чему посвятить ограниченное время своего пребывания на Земле, я все-таки отдаю предпочтение поиску других ошибок, реализации полезных возможностей, шлифовке документации и личному общению – огромному большинству пользователей это принесет куда больше пользы, чем сведение вероятности коллизии к нулю. Git полагается на хэш-коды для регистрации операций фиксации, а пользователи уже выполнили миллионы (или милларды?) таких операций, и тем не менее устранение коллизий хэша, похоже, стоит на самом последнем месте в планах людей, сопровождающих Git.

Проверка типов

Мы хотим разрешить пользователям хранить произвольные функции в хэш-таблице и поручить нашей функции диспетчеризации поиск подходящей функции и её использование предопределенным способом. Если пользователь напишет функцию, принимающую параметры неподходящих типов, то функция диспетчеризации «грохнется», и пользователь разошлет во все социальные сети саркастические комментарии о неработоспособности вашей программы.

В обычной ситуации, когда функция вызывается явно, типы параметров проверяются на этапе компиляции. С одной стороны, именно такую типобезопасность мы и теряем при динамическом выборе функции, но, с другой стороны, мы можем воспользоваться этим для проверки правильности сигнатуры функции.

Предположим, что наши функции должны принимать параметры типа `double*` и `int` (например, список или его длину) и возвращать структуру типа `out_type`. Ее тип можно тогда определить следующим образом:

```
typedef out_type (*object_fn_type)(double *, int);
```

Теперь определим ничего не делающую функцию-пустышку:

```
void object_fn_type_check(object_fn_type in){ };
```

В примере ниже она будет обернута макросом, чтобы пользователи гарантированно ее вызывали. Вызов этой функции возвращает нам типобезопасность: если пользователь попытается поместить в хэш-таблицу функцию с неправильными аргументами, то компилятор «ругнется» при попытке откомпилировать обращение к функции-пустышке.

Соберем все вместе

В примере 11.11 показан заголовок, необходимый для реализации v-таблицы. Он содержит макрос для добавления новых методов и функцию диспетчеризации, которая производит поиск в таблице.

Пример 11.11 ❖ Заголовок для v-таблицы, ассоциирующей функции с объектами (print_vtable.h)

```
#include <glib.h>
#include "print_typedef.h"
```

```
extern GHashTable *print_fns;

typedef void (*print_fn_type)(textlist_s*);      ❶

void check_print_fn(print_fn_type pf);

#define print_hash_add(object, print_fn){ \      ❷
  check_print_fn(print_fn); \
  g_hash_table_insert(print_fns, (object)->print, print_fn); \
}

void textlist_print_html(textlist_s *in);
```

❶ Это необязательно, но хороший typedef существенно облегчает работу с указателями на функции.

❷ Уговаривать пользователей пользоваться функцией проверки типа – пустая трата времени, поэтому напишем макрос, который будет это делать за них.

В примере 11.12 приведена функция диспетчеризации, которая на первом шаге производит поиск в v-таблице. Если не считать того, что просматривается v-таблица, а не сама входная структура, эта функция несильно отличается от показанного выше метода диспетчеризации.

Пример 11.12 ❖ Функция диспетчеризации, работающая с виртуальной таблицей (print_vtable.c)

```
#include <stdio.h>
#include "print_vtable.h"

GHashTable *print_fns;                                                ❶

void check_print_fn(print_fn_type pf) { }                             ❷

void textlist_print_html(textlist_s *in){
   if (!print_fns) print_fns = g_hash_table_new(g_direct_hash,        ❸
                                               g_direct_equal);

   print_fn_type ph = g_hash_table_lookup(print_fns, in->print);      ❹
   if (ph) {
      ph(in);
      return;
   }

   printf("<title>%s</title>\n<ul>", in->title);
   for (int i=0; i < in->len; i++)
      printf("<li>%s</li>\n", in->items[i]);
   printf("</ul>\n");
}
```

❶ Отметим, что хэш-таблица помещена в закрытую часть реализации, а не в открытый интерфейс. Пользователям работать с ней напрямую не придется.

❷ Из всего представленного в книге кода это моя любимая функция.

❸ Инициализируем хэш-таблицы Glib, задавая функции хэширования и сравнения на равенство. После того как они сохранены в структуре хэш-таблицы, пользователям уже не придется обращаться к ним явно. В этой строчке настраивается хэш-таблица функций печати, но при желании можно было бы настроить много других таблиц.

❹ Метод print входной структуры можно использовать для того, чтобы понять, хранится ли в структуре текст песни, кулинарный рецепт или еще что-то, так что этот метод можно использовать для поиска подходящего метода печати в виде HTML-кода.

И наконец, в примере 11.13 показано, как все это используется. Обратите внимание, что пользователь только вызывает макрос, чтобы ассоциировать специальную функцию с объектом и зарегистрировать функцию диспетчеризации, которая выполняет всю содержательную работу.

Пример 11.13 ❖ Виртуальная таблица для ассоциирования функций с объектами (print_vtable_use.c)

```
#define skip_main
#include "print_methods.c"
#include "print_vtable.h"

static void song_print_html(textlist_s *in){
  printf("<title>♫%s ♫</title>\n", in->title);
  for (int i=0; i < in->len; i++)
    printf("%s<br>\n", in->items[i]);
}

int main(){
  textlist_print_html(&save);              ❶
  printf("\n-----\n\n");
  print_hash_add(&save, song_print_html);  ❷
  textlist_print_html(&save);
}
```

❶ Сейчас хэш-таблица пуста, поэтому будет вызван метод печати по умолчанию, встроенный в функцию диспетчеризации.

❷ Здесь мы добавляем в хэш-таблицу специальный метод печати, поэтому при следующем обращении к функции диспетчеризации он будет найден и вызван.

V-таблицы – это почти официальный механизм реализации многих средств в объектно-ориентированных языках, и ничего особо сложного в нем нет. В приведенных выше примерах v-таблицам отведен едва ли десяток строчек[1]. И даже задание специальных функций для некоторых комбинаций объектов при такой

[1] Поскольку сноски никто не читает, я, пожалуй, признаюсь в любви к m4, макроязыку, созданному в 1970-е годы. Наверное, на вашем компьютере он есть, потому что он используется в Autoconf. Так вот, у этого вездесущего языка есть две уникальные и весьма полезные черты. Во-первых, он задумывался для поиска макросов в файле, написанном для совсем другой цели, например в скриптах оболочки, которые порождает

организации работает, что особенно ценно, поскольку не пришлось изобретать какой-то вычурный синтаксис для этой цели. Для реализации v-таблиц нужно немного потрудиться, но зачастую это можно отложить до более поздней версии, когда они действительно понадобятся, а на практике их преимущества проявляются лишь при выполнении некоторых операций над некоторыми структурами.

Область видимости

Область видимости переменной – это та часть программы, в которой эта переменная существует и может использоваться. Разумный программист стремится максимально ограничить область видимости любой переменной, потому что тем самым уменьшается количество переменных, о которых нужно помнить в каждый момент времени, и снижается риск, что переменная будет изменена непреднамеренно.

Засим сформулируем правила областей видимости в C.

- Переменная не видна, пока не объявлена. Поступать иначе было бы странно.
- Если переменная объявлена внутри фигурных скобок, то по достижении закрывающей скобки она выходит из области видимости. Частичное исключение: в циклах `for` и в функциях переменные могут быть объявлены внутри круглых скобок до соответствующей открывающей фигурной скобки.
- Если переменная объявлена вне фигурных скобок, то ее область видимости простирается от точки объявления до конца файла.

Вот и всё.

Не существует области видимости класса, прототипа, пространства имен, членов-друзей, перепривязки среды времени выполнения и специальных ключевых слов и операторов, относящихся к областям видимости (за исключением уже упомянутых фигурных скобок и, пожалуй, спецификаторов компоновки `static` и `extern`). Вас смущает динамичность областей видимости? Не о чем беспокоиться. Если вы знаете, где находятся фигурные скобки, то можете легко понять, какие переменные в каком месте можно использовать.

Все остальное – просто следствие. Например, если в файле *code.c* имеется строка `#include <header.h>`, то весь текст файла *header.h* включается в *code.c*, и находящиеся там переменные оказываются в соответствующей области видимости.

Функции – частный случай области видимости внутри фигурных скобок. Следующая функция суммирует все целые числа, не превосходящие значения ее аргумента:

Autoconf, или в HTML-файлах, или в программах на C. По завершении макрообработки получается стандартный скрипт оболочки, HTML-файл или исходный файл на C без каких-либо следов m4. Во-вторых, на нем можно писать макросы, порождающие другие макросы. Препроцессор C такого делать не умеет. В одном проекте, где я заранее знал, что придется генерировать множество различных v-таблиц, я написал на m4 макросы, которые генерировали функции проверки типов, и обычные C-макросы. Объем избыточности в коде резко уменьшился, а поместив в makefile шаг обработки с помощью m4, я смог распространять код на чистом C. Никто не мешает и вам применить такую предварительную обработку исходного кода, потому что m4 есть повсюду.

```
int sum (int max){
    int total=0;
    for (int i=0; i<= max; i++){
        total += i;
    }
    return total;
}
```

Переменные max и total видны внутри функции по правилу фигурных скобок с учетом частичного исключения, согласно которому переменные внутри круглых скобок до открывающей фигурной трактуются так, будто они объявлены внутри фигурных скобок. То же самое относится и к циклу for: время жизни переменной i ограничено областью внутри фигурных скобок, охватывающих тело цикла. Если тело цикла for состоит всего из одной строки, то фигурные скобки необязательны, например `for (int i=0; i <= max; i++) total += i;`, но область видимости i все равно ограничена циклом.

Подведем итог: язык С замечателен своими простыми правилами областей видимости, сводящимися, по существу, к поиску закрывающей фигурной скобки или конца файла. Объяснить всю эту систему начинающему можно минут за десять. Опытный автор может с успехом воспользоваться этими правилами для организации дополнительных областей видимости в необычных ситуациях; такие приемы описаны в разделе «Выращивание устойчивых и плодоносящих макросов» в главе 8.

Закрытые элементы структуры

Итак, мы с облегчением очистились от всех дополнительных правил и ключевых слов, поддерживающих детальное управление видимостью.

А можно ли реализовать закрытые элементы структуры без таких ключевых слов? В типичном объектно-ориентированном языке «закрытые» данные не шифруются компилятором, и вообще каких-то серьезных мер по их сокрытию никто не предпринимает: зная адрес переменной (например, ее смещение от начала структуры), мы можем создать указатель на нее, посмотреть на нее в отладчике и даже изменить. Технология для обеспечения такого ограниченного уровня закрытости у нас тоже есть.

Типичный объект определяется в двух файлах: *c*-файл содержит детали реализации, а *h*-файл включается туда, где объект используется. Почему бы не считать, что *c*-файл – это закрытая часть, а *h*-файл – открытая? Например, предположим, нам кровь из носу нужно сделать некоторые элементы объекта закрытыми. Тогда открытый заголовок мог бы выглядеть так:

```
typedef struct a_box_s {
  int public_size;
  void *private;
} a_box_s;
```

Указатель private пользователям практически бесполезен, потому что они не знают, к какому типу его приводить. А в закрытом файле *a_box.c* мы определили бы и использовали соответствующий typedef:

```
typedef struct private_box_s {
  long double how_much_i_hate_my_boss;
    char **coworkers_i_have_a_crush_on;
    double fudge_factor;
} private_box_s;
```

// Имея этот typedef, мы можем привести указатель private к истинному
// типу и воспользоваться им в файле a_box.c.

```
a_box_s *box_new(){
    a_box_s *out = malloc(sizeof(a_box_s));
    private_box_s *outp = malloc(sizeof(private_box_s));
    *out = (a_box_s){.public_size=0, .private=outp};
    return out;
}

void box_edit(a_box_s *in){
    private_box_s *pb = in->private;
    // теперь работаем с закрытыми переменными, например:
    pb->fudge_factor *= 2;
}
```

Так что совсем нетрудно реализовать закрытую часть структуры в С, но я редко встречаю такую технику в реальных библиотеках. Немногие пишущие на С считают, что это дает какое-то преимущество.

Вот гораздо более разумный способ поместить закрытый элемент в открытую структуру:

```
typedef struct {
    int pub_a, pub_b;
    int private_a, private_b; // Закрыты: просьбы не пользоваться.
} public_s;
```

Просто документируйте, что ничто не должно использоваться напрямую, и доверьтесь пользователям. Если ваши коллеги неспособны последовать такой простой инструкции, то надо приковывать кофеварку цепью к стене, потому что ваши проблемы гораздо серьезнее тех, с которыми может справиться компилятор.

Особенно просто сделать закрытыми функции: просто не помещайте их объявления в заголовок. Еще можно поставить перед определением ключевое слово `static`, чтобы пользователь уж точно знал – эта функция закрыта.

Перегрузка

У меня сложилось впечатление, что большинство программистов считает правило целочисленного деления – тот факт, что `3/2==1`, – неудобным. Когда я пишу `3/2`, то ожидаю получить `1.5`, а не единицу, черт бы ее побрал!

И это действительно досадная черта С и других языков с целочисленной арифметикой, а в более широком контексте демонстрация опасностей, которые несет *перегрузка операторов*. Так называется явление, когда действие оператора, например /, зависит от типов его операндов. Если оба операнда – целые числа, то под-

разумевается деление с отбрасыванием дробной части, во всех остальных случаях – обычное деление.

В разделе «Указатели без malloc» я говорил, что предметы, обладающие различными функциями, не должны быть похожи. В этом-то и заключается проблема `3/2`: целочисленное деление и деление с плавающей точкой ведут себя по-разному, но выглядят одинаково. А отсюда путаница и ошибки.

Естественный язык обладает избыточностью, и это хорошо, в частности потому, что помогает исправлять ошибки. Когда Нина Симон[1] поет «ne me quitte pas» (дословно это переводится «не оставляй меня не»), можно опустить как первое слово, потому что во фразе «...me quitte pas» частица *pas* обозначает отрицание, так и последнее, потому что во фразе «ne me quitte ...» отрицание обозначается словом *ne*.

Грамматический род обычно не несет большого смысла в реальных ситуациях, но иногда объект меняется в зависимости от выбора слова. Мой любимый пример – слова *el pene* и *la polla*[2] в испанском языке, которые обозначают один и тот же объект, но первое мужского рода, а второе женского. Истинная ценность рода состоит в том, что он обеспечивает избыточность за счет согласования частей предложения, а значит, делает его смысл понятнее.

Из языков программирования избыточность сознательно устраняется. Отрицание полностью меняет смысл выражения, но обычно выражается всего одним символом (`!`). Однако род в языках программирования есть, и называется он типом. В общем случае глаголы и существительные должны быть согласованы по роду (как в арабском, русском, иврите и многих других языках). При наличии такой дополнительной избыточности пришлось бы заводить функцию `matrix_multiply(a, b)` для перемножения матриц и `complex_multiply(a, b)` для перемножения комплексных чисел.

Перегрузка операторов призвана устранить избыточность, дав возможность писать `a * b` вне зависимости от того, что перемножается: матрицы, комплексные числа, натуральные числа и т. д. Приведу цитату из великолепного очерка о цене этой устраненной избыточности: «Видя в программе на C запись `i = j * 5;`, вы хотя бы знаете, что `j` умножается на пять и результат сохраняется в `i`. Но, глядя на точно такое же предложение в C++, вы не узнаете ничего»[3]. Проблема в том, что вы не знаете, что означает `*`, пока не посмотрите объявление типа `j`, не пройдете по цепочке наследования этого типа, чтобы выяснить, *какая версия* `*` имеется в виду, а потом не сделаете то же самое для `i` и не разберетесь, как оператор `=` соотносится с типами `i` и `j`.

Вот какое правило перегрузки, с помощью `_Generic` или иных средств, я вывел *для себя: если пользователь ничего не знает о входном типе, сможет ли он все-*

---

[1] Американская певица, пианистка, композитор, аранжировщица. – *Прим. перев.*

[2] Любознательному читателю не составит труда найти перевод самостоятельно. Намекну лишь, что в русском ситуация в точности аналогична, только эти слова обозначают не совсем одно и то же. – *Прим. перев.*

[3] Первоначально на странице http://www.joelonsoftware.com/articles/Wrong.html. Повторено в книге [Spolsky 2008, стр. 192].

таки прийти к правильному выводу? Например, перегрузка абсолютной величины для int, float и double согласуется с этим правилом. В библиотеке GNU Scientific Library имеется тип gsl_ complex для представления комплексных чисел, тогда как в стандартном C есть тип complex double и ему подобные; имеет смысл перегрузить функции, работающие с этими типами, так чтобы они вели себя одинаково.

Как мы видели в предыдущих примерах, C в целом придерживается описанных выше правил «согласования рода»:

```
// сложить два вектора из библиотеки GNU Scientific Library
gsl_vector *v1, *v2;
gsl_vector_add(v1, v2);

// открыть канал ввода-вывода GLib для чтения указанного файла
GError *e;
GIOChannel *f = g_io_channel_new_file("indata.csv", "r", &e);
```

Кода получается больше и когда подряд идет 10 строк, в которых упоминается одна и та же структура, создается впечатление о чрезмерном изобилии повторов, зато назначение каждой строки не вызывает никаких сомнений.

_Generic

В языке C имеется ограниченная поддержка перегрузки с помощью ключевого слова _Generic. Оно вычисляет значение в зависимости от типа входного аргумента, что позволяет писать макросы, консолидирующие несколько типов.

Не зависящие от типа функции нужны, когда типов слишком много. В некоторых системах предлагается обширный перечень точных типов, но ведь каждый из них нужно поддерживать. Например, в библиотеке GNU Scientific Library есть тип комплексного числа, комплексного вектора и просто вектора – и это при том, что в C тоже имеется тип complex. Если принять во внимание все комбинации этих четырех типов, то теоретически понадобится 16 функций. В примере 11.14 выписаны некоторые из них; если вы не из числа приверженцев комплексных векторов, то этот пример, скорее всего, покажется вам полной невнятицей, и вы захотите поскорее узнать, как его можно причесать.

Пример 11.14 ❖ Вот так делается колбаса – для тех, кого интересуют комплексные типы из библиотеки GSL (complex.c)

```
#include "cplx.h"                  //gsl_cplx_from_c99; см. ниже.
#include <gsl/gsl_blas.h>          //gsl_blas_ddot
#include <gsl/gsl_complex_math.h> //gsl_complex_mul(_real)

gsl_vector_complex *cvec_dot_gslcplx(gsl_vector_complex *v, gsl_complex x){
  gsl_vector_complex *out = gsl_vector_complex_alloc(v->size);
  for (int i=0; i< v->size; i++)
    gsl_vector_complex_set(out, i,
                 gsl_complex_mul(x, gsl_vector_complex_get(v, i)));
  return out;
}

gsl_vector_complex *vec_dot_gslcplx(gsl_vector *v, gsl_complex x){
```

```
  gsl_vector_complex *out = gsl_vector_complex_alloc(v->size);
  for (int i=0; i< v->size; i++)
    gsl_vector_complex_set(out, i,
                 gsl_complex_mul_real(x, gsl_vector_get(v, i)));
  return out;
}

gsl_vector_complex *cvec_dot_c(gsl_vector_complex *v, complex double x){
  return cvec_dot_gslcplx(v, gsl_cplx_from_c99(x));
}

gsl_vector_complex *vec_dot_c(gsl_vector *v, complex double x){
  return vec_dot_gslcplx(v, gsl_cplx_from_c99(x));
}

complex double ddot (complex double x, complex double y){return x*y;}  ❶

void gsl_vector_complex_print(gsl_vector_complex *v){
  for (int i=0; i< v->size; i++) {
    gsl_complex x = gsl_vector_complex_get(v, i);
    printf("%4g+%4gi%c", GSL_REAL(x), GSL_IMAG(x),
           i < v->size-1 ? '\t' : '\n');
  }
}
```

❶ Для перемножения чисел, имеющих встроенный в С тип `complex`, достаточно просто знака *, как и для вещественных чисел.

Чтобы это «причесать», нужен заголовок, показанный в примере 11.15. В нем ключевое слово `_Generic` используется для выбора одной из приведенных выше функций в зависимости от типа входного аргумента. Первый аргумент (*управляющее выражение*) не вычисляется, `_Generic` просто проверяет его тип и выбирает соответствующее выражение. Мы хотим выбирать функцию в зависимости от двух типов, поэтому первый макрос решает, какой – второй или третий – макрос использовать.

Пример 11.15 ❖ Использование `_Generic` для внесения порядка в хаос (cplx.h)

```
#include <complex.h> // хорошие имена для комплексных типов C
#include <gsl/gsl_vector.h> //gsl_vector_complex

gsl_vector_complex *cvec_dot_gslcplx(gsl_vector_complex *v, gsl_complex x);
gsl_vector_complex *vec_dot_gslcplx(gsl_vector *v, gsl_complex x);
gsl_vector_complex *cvec_dot_c(gsl_vector_complex *v, complex double x);
gsl_vector_complex *vec_dot_c(gsl_vector *v, complex double x);
void gsl_vector_complex_print(gsl_vector_complex *v);

#define gsl_cplx_from_c99(x) (gsl_complex){.dat= {creal(x), cimag(x)}}  ❶

complex double ddot (complex double x, complex double y);

#define dot(x,y) _Generic((x),                                      \   ❷
        gsl_vector*: dot_given_vec(y),                              \
```

```
        gsl_vector_complex*: dot_given_cplx_vec(y),              \
        default: ddot)((x),(y))

#define dot_given_vec(y) _Generic((y),                           \
        gsl_complex: vec_dot_gslcplx,                            \
        default: vec_dot_c)

#define dot_given_cplx_vec(y) _Generic((y),                      \
        gsl_complex: cvec_dot_gslcplx,                           \
        default: cvec_dot_c)
```

❶ Типы `gsl_complex` и `complex double` из стандарта C99 представляют собой массив из двух элементов типа `double`: вещественной и мнимой частей [см. руководство по GSL, а также C99 и C11 §6.2.5(13)]. Нам остается только определить подходящую структуру, а для построения ее на лету есть идеальное средство – составной литерал.

❷ При первом использовании аргумент x не вычисляется, только проверяется его тип. Это означает, что при обращении вида `dot(x++, y)` инкремент x производится лишь один раз.

В примере 11.16 жизнь снова становится прекрасной (по большей части): мы можем воспользоваться макросом `dot` для перемножения `gsl_vector` и `gsl_complex`, `gsl_vector_complex` и комплексного числа в смысле C и вычисления прочих многочисленных комбинаций. Разумеется, знать выходной тип по-прежнему необходимо, так как результатом перемножения двух скаляров является скаляр, а не вектор, поэтому порядок использования результата зависит от входных типов. Комбинаторный взрыв из-за изобилия типов – болезнь серьезная, но ключевое слово `_Generic` дает хоть какое-то лекарство.

Пример 11.16 ❖ Награда: мы можем использовать макрос `dot`, не обращая внимания (ну, почти) на типы входных аргументов (simple_cplx.c)

```
#include <stdio.h>
#include "cplx.h"

int main(){
    int complex a = 1+2I;                                              ❶
    complex double b = 2+I;
    gsl_complex c = gsl_cplx_from_c99(a);

    gsl_vector *v = gsl_vector_alloc(8);
    for (int i=0; i< v->size; i++) gsl_vector_set(v, i, i/8.);

    complex double adotb = dot(a, b);                                  ❷
    printf("(1+2i) dot (2+i):%g +%gi\n", creal(adotb), cimag(adotb)); ❸

    printf("v dot 2:\n");
    double d = 2;
    gsl_vector_complex_print(dot(v, d));

    printf("v dot (1+2i):\n");
```

```
    gsl_vector_complex *vc = dot(v, a);
    gsl_vector_complex_print(vc);

    printf("v dot (1+2i) again:\n");
    gsl_vector_complex_print(dot(v, c));
}
```

❶ Объявления, содержащие complex и const, в чем-то похожи: допустимо как complex int, так и int complex.

❷ И вот, наконец, долгожданная награда: здесь макрос dot используется четыре раза с аргументами различных типов.

❸ Это встроенные в С средства получения вещественной и мнимой частей комплексного числа.

Подсчет ссылок

Оставшуюся часть этой главы мы посвятим нескольким примерам добавления подсчета ссылок в стереотипные функции создания, копирования и освобождения. Поскольку сам по себе подсчет ссылок – не такое уж сложное дело, мы придадим изложению интерес, рассмотрев содержательные примеры с учетом реальных проблем, возникающих на практике. Но, несмотря на все интересные расширения и варианты, рабочей лошадкой, встречающейся в большинстве реальных библиотек на С, остается структура, дополненная функциями.

В первом примере мы рассмотрим небольшую библиотеку всего с одной структурой, назначение в которой – прочитать весь файл в строку. Закачать весь текст романа «Моби Дик» в строку в памяти – не проблема, но хранение тысяч копий этой строки – расточительство. Поэтому вместо копирования потенциально длинной строки данных мы будем хранить представления, в которых запоминаются только начало и конец.

Коль скоро может существовать несколько представлений строки, освобождать строку разрешается только тогда, когда не останется ни одного связанного с ней представления. Благодаря инфраструктуре объектов решить эту задачу несложно.

Во втором примере, основанном на агентах микромодели формирования групп, имеется похожая проблема: пока в группе есть члены, она должна существовать, а освобождать отведенную группе память можно только после того, как ее покинет последний член.

Пример: объект подстроки

Чтобы несколько объектов могли указывать на одну и ту же строку, мы добавим в структуру еще один элемент – счетчик ссылок. Ниже описано, как следует модифицировать четыре стереотипных элемента.

- В определение типа включается указатель на целое refs. Он инициализируется только один раз (в функции new), а все копии (созданные функцией copy) разделяют саму строку и счетчик ссылок.
- Функция new инициализирует указатель refs и присваивает *refs = 1.

- Функция copy копирует входную структуру в выходную и увеличивает счетчик ссылок на 1.
- Функция free уменьшает счетчик ссылок на 1 и, если он обращается в нуль, освобождает память, выделенную разделяемой строке.

В примере 11.17 показан заголовок *fstr.h*, в котором определены основная структура для представления участка строки и дополнительная структура для представления списка таких участков.

Пример 11.17 ❖ Открытая верхушка айсберга (fstr.h)

```
#include <stdio.h>
#include <stdlib.h>
#include <glib.h>

typedef struct {        ❶
  char *data;
  size_t start, end;
  int* refs;
} fstr_s;

fstr_s *fstr_new(char const *filename);
fstr_s *fstr_copy(fstr_s const *in, size_t start, size_t len);
void fstr_show(fstr_s const *fstr);
void fstr_free(fstr_s *in);

typedef struct {        ❷
  fstr_s **strings;
  int count;
} fstr_list;

fstr_list fstr_split (fstr_s const *in, gchar const *start_pattern);
void fstr_list_free(fstr_list in);
```

❶ Надеюсь, что эти стереотипные повторения typedef/new/copy/free стали уже привычными. Функция fstr_show очень полезна для отладки.

❷ Это вспомогательная структура, а не полноценный объект. Обратите внимание, что функция fstr_split возвращает список, а не указатель на список.

В примере 11.18 приведен код библиотеки *fstr.c*. В нем мы пользуемся имеющимися в GLib средствами для чтения текстового файла и разбора совместимых с Perl регулярных выражений. Цифры соответствуют шагам, перечисленным в начале этого раздела, чтобы вам было проще следить за тем, как поле refs используется для реализации подсчета ссылок.

Пример 11.18 ❖ Объект, представляющий подстроку (fstr.c)

```
#include "fstr.h"
#include "string_utilities.h"

fstr_s *fstr_new(char const *filename){
  fstr_s *out = malloc(sizeof(fstr_s));
  *out = (fstr_s){.start=0, .refs=malloc(sizeof(int))};
```

```
  out->data = string_from_file(filename);
  out->end = out->data ? strlen(out->data): 0;
  *out->refs = 1;                                                    ❶
  return out;
}

fstr_s *fstr_copy(fstr_s const *in, size_t start, size_t len){       ❷
  fstr_s *out = malloc(sizeof(fstr_s));
  *out=*in;
  out->start += start;
  if (in->end > out->start + len)
     out->end = out->start + len;
  (*out->refs)++;                                                    ❸
  return out;
}

void fstr_free(fstr_s *in){                                          ❹
  (*in->refs)--;
  if (!*in->refs) {
     free(in->data);
     free(in->refs);
  }
  free(in);
}

fstr_list fstr_split (fstr_s const *in, gchar const *start_pattern){
  if (!in->data) return (fstr_list){ };
  fstr_s **out=malloc(sizeof(fstr_s*));
  int outlen = 1;
  out[0] = fstr_copy(in, 0, in->end);
  GRegex *start_regex = g_regex_new (start_pattern, 0, 0, NULL);
  gint mstart=0, mend=0;
  fstr_s *remaining = fstr_copy(in, 0, in->end);
  do {                                                               ❺
     GMatchInfo *start_info;
     g_regex_match(start_regex, &remaining->data[remaining->start],
                   0, &start_info);
     g_match_info_fetch_pos(start_info, 0, &mstart, &mend);
     g_match_info_free(start_info);
     if (mend > 0 && mend < remaining->end - remaining->start){       ❻
         out = realloc(out, ++outlen * sizeof(fstr_s*));
         out[outlen-1] = fstr_copy(remaining, mend, remaining->end-mend);
         out[outlen-2]->end = remaining->start + mstart;
         remaining->start += mend;
     } else break;
  } while (1);
  fstr_free(remaining);
  g_regex_unref(start_regex);
  return (fstr_list){.strings=out, .count=outlen};
}

void fstr_list_free(fstr_list in){
  for (int i=0; i< in.count; i++){
     fstr_free(in.strings[i]);
```

```
  }
  free(in.strings);
}

void fstr_show(fstr_s const *fstr){
  printf("%.*s", (int)fstr->end-fstr->start, &fstr->data[fstr->start]);
}
```

❶ Для нового объекта fstr_s счетчик ссылок устанавливается в 1. Строки, предшествующие этой, стереотипны.
❷ Функция copy копирует переданную ей структуру fstr_s и устанавливает позиции начала и конца подстроки (проверяя, что конец не дальше конца входной структуры fstr_s).
❸ Здесь увеличивается счетчик ссылок.
❹ Здесь счетчик ссылок используется – мы решаем, нужно освобождать исходную строку или нет.
❺ В этой функции используются совместимые с Perl регулярные выражения для разбиения входной строки на участки. Как сказано в разделе «Разбор регулярных выражений» на стр. 321, сопоставитель (matcher) определяет участок строки, соответствующий переданному регулярному выражению, после чего мы с помощью fstr_copy можем получить копию этого участка. Затем мы пытаемся выделить следующий участок, выполнив те же действия с позиции, следующей за концом предыдущего участка.
❻ В противном случае соответствие не найдено, или мы дошли до конца строки.

И наконец, само приложение. Для этого нам понадобится текст романа «Моби Дик, или Белый кит» Германа Мелвилла. В примере 11.19 показано, как скачать его с с сайта проекта Гутенберг.

Пример 11.19 ❖ Используем curl для получения текста «Моби Дика» с сайта проекта Гутенберг, затем с помощью sed вырезаем начало и конец, присутствующие в любом скачанном с этого сайта файле. При необходимости установите curl с помощью менеджера пакетов (find.moby)

```
if [ ! -e moby ] ; then
  curl http://www.gutenberg.org/cache/epub/2701/pg2701.txt \
     | sed -e '1,/START OF THIS PROJECT GUTENBERG/d'       \
     | sed -e '/End of Project Gutenberg/,$d'              \
     > moby
fi
```

Получив текст книги, программа из примера 11.21 разбивает его на главы и, применяя ту же самую функцию разбиения, подсчитывает, сколько раз в каждой главе встречаются слова *whale(s)* и *I*. Обратите внимание, что в этом месте структуры fstr можно рассматривать как непрозрачные объекты, из которых используются только функции new, copy, free, show и split.

Для этой программы необходимы библиотека GLib, файл *fstr.c* и утилиты работы со строками, разработанные ранее. Таким образом, makefile выглядит, как показано в примере 11.20.

Пример 11.20 ❖ Простой makefile для программы изучения китов (cetology.make)

```
P=cetology
CFLAGS=`pkg-config --cflags glib-2.0` -g -Wall -std=gnu99 -O3
LDLIBS=`pkg-config --libs glib-2.0`
objects=fstr.o string_utilities.o

$(P): $(objects)
```

Пример 11.21 ❖ Книга разбивается на главы, и для каждой главы подсчитываются некоторые характеристики (cetology.c)

```
#include "fstr.h"

int main(){
  fstr_s *fstr = fstr_new("moby");
  fstr_list chapters = fstr_split(fstr, "\nCHAPTER");
  for (int i=0; i< chapters.count; i++){
     fstr_list for_the_title=fstr_split(chapters.strings[i],"\\.");
     fstr_show(for_the_title.strings[1]);
     fstr_list me = fstr_split(chapters.strings[i], "\\WI\\W");
     fstr_list whales = fstr_split(chapters.strings[i], "whale(s|)");
     fstr_list words = fstr_split(chapters.strings[i], "\\W");
     printf("\nch%i, words:%i.\t Is:%i\twhales:%i\n", i, words.count-1,
            me.count-1, whales.count-1);
     fstr_list_free(for_the_title);
     fstr_list_free(me);
     fstr_list_free(whales);
     fstr_list_free(words);
  }
  fstr_list_free(chapters);
  fstr_free(fstr);
}
```

Чтобы у вас был стимул попробовать эту программу, я не стану полностью воспроизводить результаты. Но сделаю несколько замечаний, из которых станет понятно, как трудно было бы Мелвиллу опубликовать или хотя бы разместить книгу в блоге в наши дни:

- длины глав различаются на порядок;
- киты почти не упоминаются вплоть до главы 30;
- рассказчик определенно обладает собственным голосом. Даже в знаменитой главе о цетологии местоимение «я» употребляется 60 раз, что придает личный характере тексту, который в противном случае можно было бы принять за статью из энциклопедии;
- имеющийся в GLib анализатор регулярных выражений оказался несколько медленнее, чем я рассчитывал.

Пример: основанная на агентах модель формирования групп

Здесь мы рассмотрим пример основанной на агентах модели членства в группе. Агенты берутся из двухмерного пространства предпочтений (поскольку мы будем представлять группы графически), а точнее, квадрата с противоположными вер-

шинами в точках (−1, −1) и (1, 1). В каждом раунде агенты будут присоединяться к группе с наибольшей полезностью. Полезность группы для агента вычисляется как −(расстояние до средней позиции группы + M*количество членов). Средняя позиция группы определяется как среднее арифметическое позиций ее членов (за исключением агента, опрашивающего группу), а константа M характеризует желание агентов оказаться в большой группе, по сравнению с их заинтересованностью в близости к средней позиции группы: если M близко к нулю, то размер группы практически безразличен, а агентов интересует только близость; если же M стремится к бесконечности, то позиция становится несущественной, а значение имеет только размер группы.

При определенном стечении обстоятельств агент формирует новую группу. Но поскольку агенты заново выбирают себе группу в каждом раунде, может случиться, что в следующем раунде агент покинет первоначально выбранную группу.

Задача подсчета ссылок похожа на ранее рассмотренную, да и вся процедура в целом аналогична:

- в определении типа имеется целочисленное поле `counter`;
- функция `new` присваивает `counter = 1`;
- функция `copy` выполняет `counter++`;
- функция `free` проверяет условие `if(--counter==0)`, и если оно истинно, то освобождает все разделяемые данные, в противном случае она оставляет все как есть, потому что еще остаются активные ссылки на структуру.

Поскольку все изменения структуры инкапсулированы в интерфейсных функциях, при работе с этим объектом думать о выделении памяти не придется.

Программа моделирования занимает почти 125 строк, а поскольку для ее документирования я пользовался CWEB, то размер чуть ли не удваивается (см. раздел «Грамотное программирование с помощью CWEB», где приведены рекомендации по поводу работы с CWEB). Благодаря выбранному стилю код должен быть совершенно понятен; даже если вы предпочитаете пропускать большие куски кода, просмотрите его хотя бы мельком. Если у вас под рукой имеется CWEB, можете сгенерировать документацию в формате PDF и почитать ее.

Предполагается, что выход этой программы будет подан на вход Gnuplot – программы построения графиков, которую исключительно легко автоматизировать. Ниже приведен командный скрипт, в котором Gnuplot передается встроенный документ, содержащий последовательность данных (e обозначает конец последовательности).

```
cat << "------" | gnuplot --persist
set xlabel "Year"
set ylabel "U.S. Presidential elections"
set yrange [0:5]
set key off
plot '-' with boxes
2000, 1
2001, 0
2002, 0
2003, 0
```

```
2004, 1
2005, 0
e
------
```

Вероятно, вы уже понимаете, как можно сформировать команды Gnuplot из программы, воспользовавшись несколькими вызовами `printf` для задания настроек и циклом `for` для вывода набора данных. Кроме того, если передать Gnuplot несколько графиков, то будут сгенерированы кадры анимации.

Приведенная далее модель порождает такую анимацию, которую можно отобразить на экране командой `./groups | gnuplot`. Напечатать результат анимации на страницах книги сложновато, так что вам придется выполнить программу самостоятельно. Вы увидите, что, хотя такое поведение не было запрограммировано, появление новых групп приводит к смещению расположенных рядом с ними, так что в итоге позиции групп оказываются равномерно распределены в пространстве. Политологи часто наблюдают подобное поведение в пространстве политических партий: когда появляются новые партии, существующие корректируют свои позиции.

Но перейдем к файлу-заголовку. То, что я называю функциями присоединения и выхода, обычно принято называть функциями копирования и освобождения. В структуре `group_s` есть поле `size`, в котором хранится количество членов группы – счетчик ссылок. Я пользуюсь библиотеками Apophenia и GLib. Отметим, что группы хранятся в связанном списке, доступном только в файле *groups.c*; для работы с этим списком нужны всего две строки кода, содержащие обращения к `g_list_append` и `g_list_remove` (пример 11.22).

Пример 11.22 ❖ Открытая часть объекта `group_s` (groups.h)

```
#include <apop.h>
#include <glib.h>

typedef struct {
  gsl_vector *position;
  int id, size;
} group_s;

group_s* group_new(gsl_vector *position);
group_s* group_join(group_s *joinme, gsl_vector *position);
void group_exit(group_s *leaveme, gsl_vector *position);
group_s* group_closest(gsl_vector *position, double mb);
void print_groups();
```

Далее приведен файл, содержащий детали реализации объекта группы.

Пример 11.23 ❖ Объект `group_s` (groups.w)

```
@ Это пролог: мы включаем заголовок и объявляем глобальный список групп,
который будет использоваться в программе. Для каждой группы нам будут нужны
функции new/copy/free.

@c
```

```
#include "groups.h"

GList *group_list;
@<new group@>
@<copy group@>
@<free group@>

@ Метод группы new стереотипный: мы выделяем память с помощью |malloc|,
заполняем структуру, пользуясь позиционными инициализаторами, и добавляем
новую группу в список.

@<new group@>=
group_s *group_new(gsl_vector *position){
  static int id=0;
  group_s *out = malloc(sizeof(group_s));
  *out = (group_s) {.position=apop_vector_copy(position), .id=id++,
                    .size=1};
  group_list = g_list_append(group_list, out);
  return out;
}

@ Когда агент присоединяется к группе, группа 'копируется', но при этом
перемещается не так уж много памяти: группа просто модифицируется для
включения еще одного человека. Мы должны увеличить счетчик ссылок (это
просто), а затем пересчитать среднюю позицию. Если средняя позиция без
учета $n$ого человека равна $P_{n-1}$, а $n$ый человек занимает позицию
$p$, то новая средняя позиция с учетом этого человека $P_n$ равна
взвешенной сумме.

$$P_n = \left( (n-1)P_{n-1}/n \right) + p/n.$$

Это выражение вычисляется для каждого измерения.

@<copy group@>=
group_s *group_join(group_s *joinme, gsl_vector *position){
  int n = ++joinme->size; // увеличить счетчик ссылок
  for (int i=0; i< joinme->position->size; i++){
     joinme->position->data[i] *= (n-1.)/n;
     joinme->position->data[i] += position->data[i]/n;
  }
  return joinme;
}

@ Функция 'free' освобождает память, выделенную группе, только когда
счетчик ссылок обращается в нуль. Если это не так, то нужно пересчитать
среднюю позицию группы после выбытия из нее члена.
@<free group@>=
void group_exit(group_s *leaveme, gsl_vector *position){
  int n = leaveme->size--; // уменьшить счетчик ссылок
  for (int i=0; i< leaveme->position->size; i++){
     leaveme->position->data[i] -= position->data[i]/n;
     leaveme->position->data[i] *= n/(n-1.);
  }
```

```
  if (leaveme->size == 0){ // убрать мусор?
     gsl_vector_free(leaveme->position);
     group_list= g_list_remove(group_list, leaveme);
     free(leaveme);
  }
}

@ Я экспериментировал с различными правилами вычисления расстояния между
агентом и группой. И в конце концов остановился на норме $L_3$. Стандартное
евклидово расстояние между точками $(x_1, y_1)$ и $(x_2, y_2)$ называется
нормой $L_2$ и вычисляется по формуле $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$.
А норма $L_3$ вычисляется по формуле $\sqrt[3]{(x_1-x_2)^3+(y_1-y_2)^3}$.
Этот вызов и вызов |apop_copy| выше - единственные обращения к функциям из
библиотеки Apophenia; если ее нет под рукой, то можно без труда написать
эти функции самостоятельно.

@<distance@>=
apop_vector_distance(g->position, position, .metric='L', .norm=3)

@ Под 'ближайшей' я понимаю группу с максимальной выгодой, для которой
величина "расстояние минус взвешенный размер" минимальна. Если дана функция
полезности, представленная кривой |dist|, то для нахождения минимального
расстояния достаточно простого цикла |for|.

@c
group_s *group_closest(gsl_vector *position, double mass_benefit){
  group_s *fave=NULL;
  double smallest_dist=GSL_POSINF;
  for (GList *gl=group_list; gl!= NULL; gl = gl->next){
     group_s *g = gl->data;
     double dist= @<distance@> - mass_benefit*g->size;
     if(dist < smallest_dist){
        smallest_dist = dist;
        fave = g;
     }
  }
  return fave;
}

@ Gnuplot легко автоматизируется. Следующая функция анимирует результаты
моделирования, для чего достаточно всего четырех строчек. Заголовок
|plot '-'| просит систему построить график по следующим далее точкам. Затем
выводятся координаты самих точек $(X, Y)$, по одной на строке. Завершающий
маркер |e| обозначает конец набора данных. Головная программа задает
начальные настройки Gnuplot.

@c
void print_groups(){
  printf("plot '-' with points pointtype 6\n");
  for (GList *gl=group_list; gl!= NULL; gl = gl->next)
     apop_vector_print(((group_s*)gl->data)->position);
  printf("e\n");
}
```

Теперь, когда у нас есть объект группы вкупе с интерфейсными функциями для добавления групп, присоединения и выхода из группы, головная программа может сосредоточиться на процедуре моделирования: определить массив людей, а затем войти в цикл проверки членства и печати (пример 11.24).

Пример 11.24 ❖ Основанная на агентах модель, в которой используется объект group_s (groupabm.w)

```
@* Инициализация.

@ Это часть основанной на агентах модели, включающая обработчики структур
|people| и саму процедуру.

Интерфейс с группами осуществляется с помощью функций new/join/exit/print из файла
|groups.cweb.c|. Поэтому в этом файле нет никакого кода для управления
памятью - механизм подсчета ссылок гарантирует, что вместе с выбытием из
группы последнего члена освобождается память, занятая самой группой.

@c
#include "groups.h"
int pop=2000,
    periods=200,
    dimension=2;

@ В функции |main| инициализируется несколько констант, которые не могут
быть сделаны статическими переменными, потому что при вычислении их
значений производятся математические вычисления.

@<set up more constants@>=
double new_group_odds = 1./pop,
       mass_benefit = .7/pop;
       gsl_rng *r = apop_rng_alloc(1234);

@* Структура |person_s|.

@ Люди в этой модели довольно скучные: они не умирают и не двигаются.
Поэтому в структуре они представлены только позицией |position| и группой,
членом которой в данный момент является агент.

@c
typedef struct {
  gsl_vector *position;
  group_s *group;
} person_s;

@ Процедура инициализации тоже скучная, ей только требуется получить
случайный вектор с равномерным распределением координат.

@c
person_s person_setup(gsl_rng *r){
  gsl_vector *posn = gsl_vector_alloc(dimension);
  for (int i=0; i< dimension; i++)
     gsl_vector_set(posn, i, 2*gsl_rng_uniform(r)-1);
```

```
  return (person_s){.position=posn};
}

@* Членство в группе.

@ В начале этой функции человек покидает свою группу. А дальше нужно принять
решение: сформировать новую группу или присоединиться к существующей.

@c
void check_membership(person_s *p, gsl_rng *r,
                      double mass_benefit, double new_group_odds){
  group_exit(p->group, p->position);
  p->group = (gsl_rng_uniform(r) < new_group_odds)
           ? @<form a new group@>
           : @<join the closest group@>;
}

@
@<form a new group@>=
group_new(p->position)

@
@<join the closest group@>=
group_join(group_closest(p->position, mass_benefit), p->position)

@* Подготовка.
@ Инициализация популяции. Благодаря макросам CWEB код самодокументирован.

@c
void init(person_s *people, int pop, gsl_rng *r){
  @<position everybody@>
  @<start with ten groups@>
  @<everybody joins a group@>
}

@
@<position everybody@>=
  for (int i=0; i< pop; i++)
     people[i] = person_setup(r);

@ Первые десять людей в списке формируют новые группы, но поскольку позиция
каждого случайна, то и положение десяти групп случайно.

@<start with ten groups@>=
  for (int i=0; i< 10; i++)
  people[i].group = group_new(people[i].position);

@
@<everybody joins a group@>=
  for (int i=10; i< pop; i++)
     people[i].group = group_join(people[i%10].group, people[i].position);

@* Построение графиков с помощью Gnuplot.

@ Это заголовок Gnuplot. Я понял, как он должен выглядеть, после того
```

```
как поэкспериментировал с Gnuplot в командной строке. Подходящие настройки
я включил сюда.

@<print the Gnuplot header@>=
printf("unset key;set xrange [-1:1]\nset yrange [-1:1]\n");

@ Анимация Gnuplot состоит из последовательносте команд plot.

@<plot one animation frame@>=
print_groups();

@* |main|.

@ Функция |main| включает несколько подготовительных шагов и простой цикл:
вычислить новое состояние и построить его график.

@c
int main(){
  @<set up more constants@>
  person_s people[pop];
  init(people, pop, r);

  @<print the Gnuplot header@>
  for (int t=0; t< periods; t++){
     for (int i=0; i< pop; i++)
        check_membership(&people[i], r, mass_benefit, new_group_odds);
     @<plot one animation frame@>
  }
}
```

Заключение

В этом разделе было приведено несколько примеров базовой формы объекта: структуры, дополненной функциями new/copy/free. Я так расщедрился на примеры, потому что несколько десятков лет развития и тысячи библиотек доказали, что это отличный способ организации кода.

В тех частях главы, где не было примеров шаблона struct/new/copy/free, продемонстрированы различные пути расширения существующих конструкций. В частности, показано, как расширить саму структуру посредством ее включения в обертывающую структуру в качестве анонимного члена.

Что касается ассоциированных функций, то мы видели несколько способов добиться того, чтобы одна функция выполняла различные действия в зависимости от переданной структуры. Если включить функции в состав структуры, то мы сможем написать функцию диспетчеризации, которая будет использовать структуру, являющуюся частью объекта. Благодаря v-таблицам функции диспетчеризации можно расширять даже после поставки структуры заказчику. Мы познакомились также с ключевым словом `_Generic`, которое позволяет решить, какую функцию вызывать, в зависимости от типа управляющего выражения.

Смогут ли все эти средства сделать вашу программу более понятной и улучшить интерфейс с пользовательским кодом, решать вам. Но вот для того чтобы

сделать более понятным *чужой код*, эти приемы определенно оказываются полезными. Возможно, вам поручено сопровождать библиотеку, написанную много лет назад, и ваши потребности вовсе не совпадают с целями ее авторов. В этом случае методы, описанные в этой главе, придутся весьма кстати: вы сможете расширить существующие структуры и добавить новые возможности в существующие функции.

Глава 12

Параллельные потоки

99 революций происходят сегодня.
— Green Day

Чуть ли не все компьютеры, проданные за последние несколько лет, – и даже многие телефоны – содержат несколько процессорных ядер. Если вы читаете этот текст на компьютере с клавиатурой и монитором, то можете узнать, сколько в нем ядер:

- Linux: `grep cores /proc/cpuinfo`;
- Mac: `sysctl hw.logicalcpu`;
- Cygwin: `env | grep NUMBER_OF_PROCESSORS`.

Однопоточная программа не полностью задействует ресурсы, которым одарил нас производитель оборудования. К счастью, не так уж сложно превратить ее в программу с параллельно выполняющимися потоками – более того, зачастую это требует всего одной дополнительной строки кода. В этой главе будут рассмотрены следующие вопросы:

- краткий обзор нескольких стандартов и спецификаций, относящихся к написанию параллельного кода на C;
- добавление вызова одной функции из библиотеки OpenMP, который сделает циклы `for` многопоточными;
- замечания о флагах компилятора, необходимых для компоновки программы с библиотекой OpenMP или pthreads;
- некоторые соображения о том, когда использование одной волшебной строки безопасно;
- реализация каркаса map-reduce, для чего, помимо вышеупомянутой строки, требуется еще кое-что;
- синтаксическая конструкция для параллельного запуска нескольких различных задач, например основной работы и ее графического интерфейса пользователя;
- ключевое слово `_Thread_local`, которое позволяет создавать поточно-локальные копии глобальных статических переменных;
- критические области и мьютексы;
- атомарные переменные в OpenMP;
- замечание о последовательной непротиворечивости и о том, зачем она нужна;

- потоки, описанные в стандарте POSIX, и их отличия от OpenMP;
- реализация атомарных скалярных переменных с помощью атомов C;
- реализация атомарных структур с помощью атомов C.

Это еще одна глава, в которой излагается материал, отсутствующий в стандартных учебниках C. Обследуя рынок, я не нашел ни одной книги общего характера по C, в которой рассматривалась бы библиотека OpenMP. Поэтому я расскажу достаточно, чтобы вы могли приступить к работе, – быть может, даже настолько много, что вам вообще не придется обращаться к книгам, специально посвященным теоретическим вопросам многопоточности.

Однако есть книги на тему параллельного программирования, в том числе на C, в которых рассматриваются многочисленные детали, для которых мне не хватает места, например [Breshears 2009], [Gove 2010], [Grama 2003]. Я буду придерживаться алгоритма планирования, подразумеваемого по умолчанию, и самой безопасной формы синхронизации, хотя в некоторых случаях можно было бы добиться большего эффекта за счет более точной настройки этих вещей. Я не стану вдаваться в детали оптимизации кэша и не приведу полного перечня полезных прагм OpenMP (его вы без труда сможете найти в Интенете).

Окружение

Механизм многопоточности вошел в стандарт C только в декабре 2011 года. Это произошло с явным запозданием, к этому моменту уже существовали решения от различных производителей. Итак, к нашим услугам несколько вариантов.

- Потоки POSIX. Стандарт pthreads был определен в POSIX v1 в 1995 году. Функция `pthread_create` сопоставляет каждому потоку функцию определенного вида, поэтому нам предстоит написать подходящий функциональный интерфейс и обычно сопровождающую его структуру.
- В Windows имеется собственная система потоков, работающая по аналогии с pthreads. Например, функция `CreateThread` принимает функцию и указатель на параметры – так же, как `pthread_create`.
- OpenMP – это спецификация, в которой используются многочисленные директивы `#pragma` и ряд библиотечных функций, с помощью которых программа сообщает компилятору, когда и как создавать поток. Именно таким способом можно превратить последовательный цикл `for` в распараллеленный, добавив всего одну строчку кода. Первая версия спецификации OpenMP для C вышла в 1998 году.
- Спецификация стандартной библиотеки C теперь включает заголовки, в которых определены функции для многопоточной работы и операций с атомарными переменными.

Что из перечисленного использовать, зависит от целевого окружения, а также ваших собственных устремлений и предпочтений. Если ваша цель – обеспечить надежную компиляцию кода как можно большим числом компиляторов, то в настоящее время, как ни странно, оптимальным будет решение, дальше других вы-

ходящее за пределы стандартного C: OpenMP. Оно поддерживается всеми основными компиляторами, даже Visual Studio[1]. На момент написания этой книги мало найдется пользователей, у которых есть компиляторы и стандартные библиотеки, поддерживающие многопоточность, описанную в стандарте C. Если вы не любите полагаться на прагмы, то воспользуйтесь библиотекой pthread – она существует для любой POSIX-совместимой платформы (даже MinGW).

Существуют и другие возможности, например MPI (интерфейс передачи сообщений, предназначенный для взаимодействия узлов сети) или OpenCL (особенно полезно для обработки на графических процессорах). В POSIX-совместимых системах можно воспользоваться системным вызовом `fork`, чтобы создать два экземпляра программы, которые разделяют общую память, но в остальном работают независимо.

Составные части

Наши синтаксические потребности скромны. Во всех случаях нам нужно следующее.

- Средства, позволяющие сообщить компилятору о необходимости запуска нескольких потоков сразу. В качестве раннего примера приведу примечание в строке 404 романа Набокова «Бледное пламя» [Nabokov 1962]: [*Тут время начало двоиться*]. Далее повествование колеблется между двумя потоками.
- Средства, позволяющие отметить точку, где дополнительные потоки прекращают существование, а главный поток продолжает работать в одиночестве. В некоторых примерах, в частности вышеупомянутом раннем примере, барьер неявно устанавливается в конце некоторого участка, тогда как в других существует явный шаг «ожидания потоков».
- Средства, позволяющие сказать, что некоторый код не должен распараллеливаться, потому что его нельзя сделать потокобезопасным. Например, что произойдет, если один поток установит новый размер массива 20, а в то же время другой поток установит размер того же массива равным 30? Конечно, изменение размера занимает всего какую-нибудь микросекунду, но если бы мы могли замедлить время, то увидели бы, что даже простейшая операция инкремента `x++` распадается в последовательность конечных операций, на которых возможен конфликт потоков. Прагмы OpenMP позволяют пометить такие нераспараллеливаемые сегменты как *критические области*, а в системах на основе pthread синхронизация обеспечивается *мьютексами* (слово *mutex* образовано в результате объединения двух слов *mutual exclusion* – взаимоисключение).
- Средства для работы с переменными, к которым одновременно могут обращаться несколько потоков. Существуют различные стратегии, например превратить глобальную переменную в поточно-локальную или синхронизировать доступ к каждой такой переменной с помощью мьютекса.

---

[1] Visual Studio поддерживает версию 2.0, а текущая версия OpenMP имеет номер 4.0, но основные прагмы, рассматриваемые в этой книге, не новы.

OpenMP

В качестве примера распараллелим программу подсчета слов. Некоторыми служебными функциями для работы со строками, позаимствованными из примера 11.21, я воспользуюсь в функции подсчета слов. Чтобы отделить ее от частей, относящихся к многопоточности, я помещу эту функцию в отдельный файл; см. пример 12.1.

Пример 12.1 ❖ **Функция подсчета слова считывает весь файл в память, а затем разбивает строку по символам, не являющимся частью слова (wordcount.c)**

```
#include "string_utilities.h"

int wc(char *docname){
   char *doc = string_from_file(docname);                    ❶
   if (!doc) return 0;
   char *delimiters = " `~!@#$%^&*()_-+={[]}|\\;:\",<>./?\n";
   ok_array *words = ok_array_new(doc, delimiters);          ❷
   if (!words) return 0;
   double out= words->length;
   ok_array_free(words);
   return out;
}
```

❶ Функция `string_from_file`, заимствованная из примера программы «изучения китов», считывает весь документ в строку в памяти.

❷ Эта функция, заимствованная из библиотеки служебных функций для работы со строками, разбивает строку по заданным разделителям. Нам от нее нужен только счетчик.

В примере 12.2 эта функция вызывается для каждого файла, указанного в командной строке. Функция `main` сводится к многократному вызову `wc` в цикле `for` и последующему суммированию отдельных счетчиков для получения общего итога.

Пример 12.2 ❖ **Добавив всего одну строку кода, мы можем выполнять различные итерации цикла `for` в разных потоках (openmp_wc.c)**

```
#include "stopif.h"
#include "wordcount.c"

int main(int argc, char **argv){
   argc--;
   argv++;                                 ❶
   Stopif(!argc, return 0, "Необходимо указать хотя бы одно имя файла в "
                           "командной строке.");
   int count[argc];
   #pragma omp parallel for                ❷
   for (int i=0; i< argc; i++){
     count[i] = wc(argv[i]);
     printf("%s:\t%i\n", argv[i], count[i]);
   }

   long int sum=0;
   for (int i=0; i< argc; i++) sum+=count[i];
     printf("Σ:\t%li\n", sum);
}
```

❶ argv[0] содержит имя программы, поэтому пропускаем его. Остальные элементы массива argv – это аргументы, заданные в командной строке, то есть файлы, для которых нужно подсчитать слова.
❷ В результате добавления этой строки цикл for распараллеливается.

Обратили внимание на строку, которая превращает эту программу в многопоточную? Она содержит инструкцию OpenMP:

```
#pragma omp parallel for
```

означающую, что следующий сразу после нее цикл for следует разбить на участки и создать для их выполнения столько потоков, сколько компилятор сочтет оптимальным. В данном случае я выполнил свое обещание превратить последовательную программу в параллельную добавлением всего одной строчки кода.

OpenMP определяет, сколько потоков может одновременно работать в системе, и соответственно разбивает работу на части. Если вы хотите задать количество потоков вручную, то нужно либо присвоить значение переменной окружения до запуска программы:

```
export OMP_NUM_THREADS=N
```

либо обратиться к библиотечной функции в самой программе:

```
#include <omp.h>

omp_set_num_threads(N);
```

Пожалуй, наиболее полезны эти средства, когда нужно задать число потоков равным 1. Чтобы вернуться в режим по умолчанию – запрашивать столько потоков, сколько имеется процессорных ядер, нужно поступить так:

```
#include <omp.h>

omp_set_num_threads(omp_get_num_procs());
```



Макрос, определенный с помощью директивы #define, не может расширяться в #pragma, и что же делать, если хочется распараллелить макрос? Для этого в стандарт введен оператор _Pragma (C99 и C11 §6.10.9). Его единственный операнд читается из строки (на жаргоне официального стандарта *destringized*) и интерпретируется как прагма. Например:

```
#include <stdio.h>

#define pfor(...) _Pragma("omp parallel for") for(__VA_ARGS__)

int main(){
pfor(int i=0; i< 1000; i++){
   printf("%i\n", i);
}
}
```

Компиляция для использования OpenMP

Для того чтобы gcc и clang (отметим, что поддержка OpenMP в clang на некоторых платформах еще реализована не полностью) откомпилировали эту программу, не-

обходимо задать флаг компилятора -fopenmp. Если требуется отдельный шаг компоновки, то этот флаг нужно задать также для компоновщика (компилятор знает, нужно ли прикомпоновывать какие-нибудь библиотеки, и сам сделает все необходимое). Для использования библиотеки pthread требуется флаг -pthread, так что если вам нужно то и другое, поместите такие строки в makefile:

```
CFLAGS=-g -Wall -O3 -fopenmp -pthread
LDLIBS=-fopenmp
```

Если вы пользуетесь Autoconf, то добавьте в файл configure.ac строку:

```
AC_OPENMP
```

Она порождает переменную $OPENMP_CFLAGS, которую затем нужно будет добавить в состав флагов в файле Makefile.am. Например:

```
AM_CFLAGS = $(OPENMP_CFLAGS) -g -Wall -O3 …
AM_LDFLAGS = $(OPENMP_CFLAGS) $(SQLITE_LDFLAGS) $(MYSQL_LDFLAGS)
```

Итого понадобилось три строчки кода, зато теперь Autoconf будет правильно компилировать программу на любой известной ему платформе, которая поддерживает OpenMP.

Стандарт OpenMP требует, чтобы в случае, когда компилятор поддерживает прагмы OpenMP, был определен символ _OPENMP. Поэтому для условной компиляции частей программы можно воспользоваться директивой #ifdef _OPENMP.

Откомпилировав программу threaded_wc, попробуйте выполнить команду ./threaded_wc `find ~ -type f` для подсчета слов во всех файлах, находящихся в домашнем каталоге. Можете запустить в другом окне команду top и посмотреть, сколько работает экземпляров wc.

Интерференция

Итак, синтаксис для превращения программы в многопоточную у нас имеется, но есть ли гарантия, что такая программа будет работать правильно? В простых случаях, когда можно доказать, что все итерации цикла независимы, да. Но есть и другие ситуации, когда нужно проявлять осторожность.

Чтобы доказать, что группа потоков работает правильно, мы должны знать, что происходит с каждой переменной и каковы побочные эффекты.

- Если переменная не видна за пределами потока, то есть уверенность, что она ведет себя так же, как в однопоточной программе, и никакой интерференции не возникает. В частности, итератор цикла, который в примере выше называется i, локален в каждом потоке (OpenMP 4.0 §2.6). Переменные, объявленные внутри цикла, видны только в этом цикле.
- Если переменная читается различными потоками, но ни один из них нигде не изменяет ее, то безопасность обеспечена. И чтобы понять это, не нужно знать высшую математику: при чтении переменной ее состояние не изменяется (я не говорю об атомарных флагах C, которые невозможно прочитать без установки).

- Если в переменную пишет один поток, а никакой другой не читает, то конкуренции опять же нет, и можно считать, что переменная по существу поточно-локальна.
- Если переменную сообща используют несколько потоков, если она записывается в одном потоке, а читается (или записывается) в другом, то возникают проблемы, и оставшаяся часть этой главы посвящена их решению.

Отсюда первое следствие: по возможности избегать разделения переменных, в которые производится запись. Один из способов добиться этого показан в примере выше: все потоки пользуются массивом `count`, но на `i`-й итерации изменяется только `i`-й элемент массива, а это значит, что каждый элемент массива можно считать поточно-локальным. Кроме того, сам массив `count` не изменяется во время выполнения цикла – его размер остается постоянным, отведенная ему память не освобождается и т. п., – и то же самое относится к массиву `argv`. А ниже мы и вовсе избавимся от массива `count`.

Мы ничего не знаем о внутренних переменных `printf`, но стандарт C требует, чтобы все функции стандартной библиотеки, работающие с потоками ввода-вывода (то есть почти все объявленные в заголовке `stdio.h`), были потокобезопасными, поэтому мы можем вызывать `printf`, не опасаясь интерференции различных вызовов (C11 §7.21.2(7) и (8)).

При написании этого примера пришлось внимательно следить за соблюдением описанных условий. Однако некоторые рекомендации, в частности совет избегать глобальных переменных, в равной мере относятся и к однопоточным программам. Кроме того, начинает приносить плоды стиль объявления переменных в точке первого использования, распространившийся после принятия стандарта C99: ведь переменная, объявленная внутри участка, который будет подвергнут распараллеливанию, безусловно, должна быть поточно-локальной.

Попутно отметим, что программа `omp parallel for` понимает только простые циклы: итератор должен иметь целый тип, увеличиваться или уменьшаться на каждой итерации на постоянную величину (инвариант цикла), и в условии окончания итератор должен сравниваться со значением инварианта цикла или с переменной. Все циклы, в которых одни и те же действия применяются к каждому элементу фиксированного массива, попадают в эту категорию.

Map-reduce

Программа подсчета слов имеет типичную структуру: каждый поток выполняет какую-то независимую задачу и порождает некоторый результат, но интерес представляет сведение этих частичных результатов в единственный агрегат. OpenMP поддерживает такую последовательность операций распределения и редукции (map-reduce) за счет добавления к рассмотренной выше прагме. В примере 12.3 массив `count` заменен одной переменной `total_wc`, а в прагму OpenMP добавлена часть `reduction(+:total_wc)`. Начиная с этого момента, компилятор сам эффективно организует работу по объединению вычисленных каждым потоком частичных значений `total_wc` в одно итоговое значение.

Пример 12.3 ❖ Чтобы реализовать в цикле `for` последовательность операций распределения и редукции, необходимо добавить фразу в прагму `#pragma omp parallel for` (mapreduce_wc.c)

```
#include "stopif.h"
#include "wordcount.c"

int main(int argc, char **argv){
   argc--;
   argv++;
   Stopif(!argc, return 0, "Необходимо указать хотя бы одно имя файла в "
                           "командной строке.");
   long int total_wc = 0;
   #pragma omp parallel for \
    reduction(+:total_wc)        ❶
   for (int i=0; i< argc; i++){
      long int this_count = wc(argv[i]);
      total_wc += this_count;
      printf("%s:\t%li\n", argv[i], this_count);
   }
   printf("Σ:\t%li\n", total_wc);
}
```

❶ Добавление фразы `reduction` в прагму `omp parallel for` сообщает компилятору, что эта переменная должна содержать сумму значений, вычисленных всеми потоками.

Но и на этот раз имеются ограничения: на месте оператора + во фразе `reduction(+:variable)` можно использовать только один из основных арифметических операторов (+, *, -), поразрядные операторы (&, |, ^) и логические операторы (&&, ||). В любом другом случае придется вернуться к чему-то наподобие массива `count` выше и самостоятельно запрограммировать процедуру редукции, выполняемую после завершения всех потоков (см. вычисление максимума в примере 12.5). Кроме того, не забудьте инициализировать редуцируемую переменную до начала работы группы потоков.

Несколько задач

До сих пор мы рассматривали применение одной и той же операции к каждому элементу массива. Но, возможно, имеются две совершенно разные операции, которые не зависят друг от друга и могут работать параллельно. Например, в программах с графическим интерфейсом пользователя (ГИП) интерфейс часто работает в одном потоке, а фоновая обработка – в другом, чтобы избежать зависания интерфейса. В этой ситуации применяется прагма `parallel sections`:

```
#pragma omp parallel sections
{
   #pragma omp section
   {
      // Все находящееся в этом блоке выполняется в одном потоке
      UI_starting_fn();
```

```
    }
    #pragma omp section
    {
       // А все находящееся в этом блоке – в другом потоке
       backend_starting_fn();
    }
}
```

В OpenMP есть и другие возможности, которые я здесь подробно рассматривать не буду, но вам они могут понравиться.

- simd: Single instruction, multiple data (одиночный поток команд, множественный поток данных). В некоторых процессорах имеется возможность применять одну и ту же операцию к каждому элементу вектора. Это не то же самое, что несколько потоков, исполняемых несколькими ядрами, и не все процессоры такой режим поддерживают. См. `#pragma omp simd`, а также руководство по компилятору, потому что некоторые компиляторы автоматически вставляют SIMD-команды там, где это возможно.
- Если количество задач заранее неизвестно, то можно воспользоваться прагмой `#pragma omp task`, которая запускает новый поток. Например, для обхода дерева можно завести один поток, а в каждом листовом узле запускать с помощью этой прагмы новый поток для его обработки.
- Можно создать несколько потоков для поиска чего-то. Но как только один поток обнаружит искомое, работа остальных потоков становится бессмысленной. Тогда, чтобы отменить менее удачливые потоки, можно воспользоваться прагмой `#pragma omp cancel` (в pthread имеется эквивалент: `pthread_cancel`).

Хочу, однако, предостеречь читателей от употребления `#pragma` перед каждым циклом `for` в программе: с созданием потоков сопряжены накладные расходы. Такой код:

```
int x = 0;
#pragma omp parallel for reduction(+:x)
for (int i=0; i< 10; i++){
  x++;
}
```

затратит больше времени на порождение потоков, чем на инкрементирование `x`, и последовательная версия почти наверняка будет работать быстрее параллельной. Не скупитесь на вкрапления многопоточности, но всякий раз проверяйте, действительно ли производительность повысилась в результате изменений.

Из того, что на создание и уничтожение потоков тратится дополнительное время, вытекает эвристическое правило: чем меньше потоков создается, тем лучше. Например, если имеется два вложенных цикла, то обычно эффективнее распараллелить внешний цикл, чем внутренний.

Если вы уверены, что ни в одном из распараллеленных участков не производится запись в разделяемую переменную и если все вызываемые функции потокобезопасны, то можете дальше не читать. Расставьте в подходящих местах прагмы `#pragma omp parallel for` или `parallel sections` и радуйтесь тому, насколько быстрее

стала работать программа. А оставшаяся часть этой главы, да и большинство работ по многопоточному программированию посвящены стратегиям модификации разделяемых ресурсов.

Поточная локальность

Статические переменные – даже объявленные внутри участка `#pragma omp parallel` – по умолчанию разделяются всеми потоками. Чтобы создать отдельную копию переменной в каждом потоке, нужно добавить в эту прагму фразу `threadprivate`, например:

```
static int state;
#pragma omp parallel for threadprivate(state)
for (int i=0; i< 100; i++)
```

При соблюдении некоторых оговорок, продиктованных здравым смыслом, система сохраняет набор поточно-локальных переменных, так что если переменная `static_x` была равна 2.7 в потоке 4 в конце одного распараллеленного участка, то в потоке 4 она будет равна 2.7 и в начале следующего участка, распараллеленного между четырьмя потоками (OpenMP §2.14.2). В любой момент существует один главный поток; вне распараллеленного участка главный поток сохраняет свою копию статической переменной.

Ключевое слово C `_Thread_local` «расщепляет» статические переменные аналогичным образом. В C для поточно-локальной статической переменной «временем жизни является все время выполнения потока, в котором она создана, а ее значение инициализируется в момент запуска потока» (C11 §6.2.4(4)). Если считать, что поток 4 в одном распараллеленном участке – то же самое, что поток 4 в другом распараллеленном участке, то это поведение ничем не отличается от специфицированного в OpenMP; если считать их разными потоками, то стандарт C следует интерпретировать так, что поточно-локальная память заново инициализируется в каждом распараллеленном участке.

По-прежнему существует главный поток, который продолжает работать вне любого распараллеленного участка (явно это не оговорено, но вытекает из C11 §5.1.2.4(1)), поэтому поточно-локальная статическая переменная в главном потоке очень похожа на традиционную статическую переменную, время жизни которой совпадает со временем работы программы.

В GCC и в clang имеется ключевое слово `__thread`, которое было принято в качестве расширения GCC еще до добавления в стандарт ключевого слова `_Thread_local`. Внутри функции допустима любая форма:

```
static __thread int i;          // специфика GCC/clang; работает сегодня
// или
static _Thread_local int i;     // C11, когда компилятор начнет поддерживать
```

Вне функции ключевое слово `static` необязательно, так как подразумевается по умолчанию. Стандарт требует, чтобы в заголовке `threads.h` был определен символ

thread_local, являющийся псевдонимом _Thread_local, точно так же, как в заголовке stdbool.h должен быть определен псевдоним bool для _Bool.

Проверить, что именно использовать, можно с помощью последовательности директив препроцессора, в которой устанавливается подходящее значение символа threadlocal. Пример такой последовательности приведен ниже:

```
#undef threadlocal
#if __STDC_VERSION__ > 201100L
  #define threadlocal _Thread_local
#elif defined(__APPLE__)
  #define threadlocal       //на момент написания этой книги не реализовано
#elif (defined(__GNUC__) || defined(__clang__)) && !defined(threadlocal)
  #define threadlocal __thread
#else
  #define threadlocal
#endif
```

Локализация нестатических переменных

Если переменная расщепляется на поточно-локальные копии, то нужно решить, как она будет инициализироваться в каждом потоке и что делать при выходе из распараллеленного участка. Фраза threadprivate() говорит OpenMP, что статическую переменную нужно инициализировать начальным значением переменной и сохранять копии при выходе из распараллеленного участка для использования при следующем входе в него.

Мы уже видели подобную фразу: reduction(+:var) требует от OpenMP инициализировать копию переменной в каждом потоке нулем (или единицей в случае умножения), дать каждому потоку возможность самостоятельно производить сложения и вычитания, а после выхода прибавить значение поточно-локальной копии к исходному значению var.

Нестатические переменные, объявленные вне распараллеленного участка, являются по умолчанию разделяемыми. Чтобы создать локальные копии переменной localvar в каждом потоке, добавьте фразу firstprivate(localvar) в строку #pragma omp parallel. Копия создается в каждом потоке и инициализируется значением переменной в момент запуска потока. По завершении потока все копии уничтожаются, а исходная переменная не изменяется. Добавьте фразу lastprivate(localvar), чтобы скопировать окончательное значение переменной в последнем потоке (том, у которого наибольший индекс в цикле for, или последнем в списке участков section) в переменную, находящуюся вне распараллеленного участка. Довольно часто можно увидеть одну и ту же переменную в обеих фразах firstprivate и lastprivate.

Разделяемые ресурсы

До сих пор я подчеркивал важность использования поточно-локальных переменных и описывал средства расщепления одной статической переменной на несколько поточно-локальных. Но бывает так, что ресурс обязан быть разделяемым, и тогда простейшим средством его защиты является *критическая область*. Так на-

зывается участок программы, который в каждый момент времени может выполняться только одним потоком. Как и большинство прагм OpenMP, эта воздействует на блок, расположенный сразу за ней:

```
#pragma omp critical (a_private_block)
{
  //здесь находится интересующий нас код
}
```

Гарантируется, что в этом блоке может находиться не более одного потока. Если некий поток подойдет к этому месту, когда в критической области находится другой поток, то он будет ждать перед входом в область, пока исполняющийся в данный момент поток не покинет ее.

Это называется *блокировкой*; блокированный поток в течение некоторого времени ничего не делает. Конечно, это неэффективно, но неэффективная программа все же гораздо лучше некорректной.

Имя (a_private_block) в скобках позволяет связать критические области между собой, например чтобы защитить один и тот же ресурс, используемый в разных местах программы. Если вы не хотите, чтобы некая структура читалась в тот момент, когда другой поток производит запись в нее, то можете воспользоваться этой возможностью:

```
#pragma omp critical (delicate_struct_region)
{
  delicate_struct_update(ds);
}

[какой-то код]

#pragma omp critical (delicate_struct_region)
{
  delicate_struct_read(ds);
}
```

Гарантируется, что в объединенной критической области, состоящей из двух участков, в каждый момент времени находится не более одного потока, и потому обращение к delicate_struct_update никогда не будет произведено одновременно с обращением к delicate_struct_read. Код между этими участками будет работать, как обычно.



Строго говоря, имя необязательно, но все неименованные критические области считаются частями одной и той же группы. Это типично для небольших программ-примеров (в том числе и тех, что можно найти в Интернете), но в нетривиальном коде может оказаться недостаточно. Присваивая каждой критической области имя, мы предотвращаем непреднамеренное связывание двух участков.

Рассмотрим задачу о нахождении количества делителей числа (простых и составных). Например, число 18 нацело делится на шесть положительных целых чисел: 1, 2, 3, 6, 9, 18. У числа 13 всего два делителя, 1 и 13, то есть это число простое.

Найти простые числа нетрудно – в диапазоне от 1 до десяти миллионов таковых 664 579. Но существует всего 446 чисел, меньших десяти миллионов, у кото-

рых ровно три делителя, шесть – имеющих ровно семь делителей и только одно с 17 делителями. Числа с другим количеством делителей более распространены; так, существует 2 228 418, меньших десяти миллионов, у которых ровно восемь делителей.

В примере 12.4 приведена программа для нахождения количества делителей, распараллеленная средствами OpenMP. В ней используются два массива. Первый, `factor_ct`, содержит десять миллионов элементов. Все элементы, кроме первого, инициализируются значением 2, потому что любое число делится на единицу и на себя. Затем мы прибавляем 1 к каждому элементу массива, индекс которого делится на два (то есть для каждого четного числа). Затем прибавляем 1 к элементам массива с индексом, кратным трем, и т. д. до пяти миллионов (в одной группе с пятимиллионным элементом оказался бы только десятимиллионный, если бы таковой был в массиве). По завершении этой процедуры мы будем знать, сколько делителей у каждого числа. При желании можете добавить цикл `for` для вывода всего массива в файл с помощью `fprintf`.

Далее создается еще один массив, в котором регистрируется, сколько чисел имеют 1, 2, ... делителей. Но перед этим мы должны найти максимальное количество делителей, чтобы знать размер массива, а уже затем можно просматривать массив `factor_ct` и заниматься подсчетами.

Каждый шаг является очевидным кандидатом на распараллеливание с помощью прагмы `#pragma omp parallel for`, но при этом могут возникнуть конфликты. Не исключено, что поток, помечающий кратные 5, и поток, помечающий кратные 7, одновременно захотят увеличить элемент `factor_ct[35]`. Чтобы предотвратить конфликт при записи, сделаем критической областью строку, в которой количество делителей элемента `i` увеличивается на единицу:

```
#pragma omp critical (factor)
factor_ct[i]++;
```



Действие этих прагм распространяется на блок, следующий прямо за ними. Блоки обычно обозначаются фигурными скобками, но если фигурных скобок нет, то одиночная строка считается отдельным блоком.

Когда один поток захочет увеличить элемент `factor_ct[30]`, он заблокирует другой поток, желающий увеличить `factor_ct[33]`. Критические области – это блоки кода, они имеют смысл, если разные блоки ассоциированы с одним и тем же ресурсом. Но в данном случае мы пытаемся защитить десять миллионов ресурсов, и это подводит нас к идее *мьютексов* и *атомарных переменных*.

Мьютекс (сокращение от *mutual exclusion*) применяется, чтобы заблокировать поток, и в этом он похож на критические области, состоящие из нескольких участков. Однако мьютекс – это обычная структура, и никто не мешает завести десять миллионов мьютексов. Если захватить `i` до записи в элемент `i` и освободить его после записи, то мы получим критическую область, защищающую только элемент `i`. В программе это выглядело бы так:

```
omp_lock_t locks[1e7];
for (long int i=0; i< lock_ct; i++)
  omp_init_lock(&locks[i]);

#pragma omp parallel for
for (long int scale=2; scale*i < max; scale++) {
  omp_set_lock(&locks[scale*i]);
  factor_ct[scale*i]++;
  omp_unset_lock(&locks[scale*i]);
}
```

Функция `omp_set_lock` на самом деле относится к типу «ждать, затем установить»: если мьютекс никем не захвачен, то она захватывает его и возвращает управление; если же мьютекс уже захвачен, то она блокирует поток и ждет, пока другой поток не выполнит функцию `omp_unset_lock`, сообщив тем самым, что путь свободен.

Мы, как и хотели, сгенерировали десять миллионов критических областей. Есть, правда, мелкая проблема – структура мьютекса сама занимает место, и для выделения десяти миллионов таких структур может уйти больше времени, чем на сами вычисления, совсем простые. Поэтому я решил использовать только 128 мьютексов и захватывать мьютекс с индексом `i% 128`. Тогда вероятность того, что два потока, работающие с разными числами, без необходимости заблокируют друг друга, составляет 1 к 128. Это не так страшно, зато на моих тестовых машинах скорость работы программы резко возросла за счет уменьшения накладных расходов на создание и использование десяти миллионов мьютексов.

Прагмы встроены в компилятор, который их понимает, но мьютексы – обычные структуры и функции, написанные на С, поэтому в программу нужно включить директиву `#include <omp.h>`. Исходный код приведен в примере 12.4, а часть, относящаяся к поиску наибольшего числа делителей, вынесена в отдельный листинг.

Пример 12.4 ❖ **Сгенерировать массив для хранения числа делителей, найти в этом массиве наибольший элемент и подсчитать, сколько есть чисел с 1, 2, … делителями (openmp_factors.c)**

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h> //malloc
#include <string.h> //memset

#include "openmp_getmax.c"                                    ❶

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    int lock_ct = 128;
    omp_lock_t locks[lock_ct];
    for (long int i=0; i< lock_ct; i++)
       omp_init_lock(&locks[i]);

    factor_ct[0] = 0;                                         ❷
    factor_ct[1] = 1;
```

```
    for (long int i=2; i< max; i++)
        factor_ct[i] = 2;

    #pragma omp parallel for
    for (long int i=2; i<= max/2; i++)
        for (long int scale=2; scale*i < max; scale++) {
            omp_set_lock(&locks[scale*i% lock_ct]);         ❸
            factor_ct[scale*i]++;
            omp_unset_lock(&locks[scale*i% lock_ct]);       ❹
        }

    int max_factors = get_max(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        int factors = factor_ct[i];
        omp_set_lock(&locks[factors% lock_ct]);             ❺
        tally[factors]++;
        omp_unset_lock(&locks[factors% lock_ct]);
    }

    for (int i=0; i<=max_factors; i++)
        printf("%i\t%li\n", i, tally[i]);
}
```

❶ См. следующий листинг.
❷ Инициализировать. Числа 0 и 1 считаем составными.
❸ Захватить мьютекс перед чтением или записью переменной.
❹ Освободить мьютекс после чтения или записи переменной.
❺ Я повторно использую имеющийся набор мьютексов, чтобы не проводить инициализацию еще раз, но здесь те же мьютексы применяются совсем для другой цели.

В примере 12.5 мы ищем максимальное значение в массиве `factor_ct`. Поскольку OpenMP не предоставляет редуктора `max`, придется завести массив, в котором каждый поток будет хранить свой локальный максимум, а затем найти в нем глобальный максимум. Длина этого массива равна `omp_get_max_threads()`, а найти свой индекс каждый поток может с помощью функции `omp_get_thread_num()`.

Пример 12.5 ❖ Распараллеленный поиск максимального элемента в массиве (openmp_getmax.c)

```
int get_max(int *array, long int max){
    int thread_ct = omp_get_max_threads();
    int maxes[thread_ct];
    memset(maxes, 0, sizeof(int)*thread_ct);

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
        int this_thread = omp_get_thread_num();
```

```
      if (array[i] > maxes[this_thread])
          maxes[this_thread] = array[i];
   }

   int global_max=0;
   for (int i=0; i< thread_ct; i++)
      if (maxes[i] > global_max)
          global_max = maxes[i];
   return global_max;
}
```

В приведенных выше примерах каждый мьютекс защищает один участок кода, но, как и в случае критических областей, можно было бы с помощью единственного мьютекса защитить ресурс, используемый в нескольких местах программы.

```
omp_set_lock(&delicate_lock);
delicate_struct_update(ds);
omp_unset_lock(&delicate_lock);

[какой-то код]

omp_set_lock(&delicate_lock);
delicate_struct_read(ds);
omp_unset_lock(&delicate_lock);
```

Атомы

Атом – это мельчайший неделимый элемент[1]. Атомарные операции часто требуют специальных возможностей процессора, и в OpenMP они ограничены только скалярами: почти всегда это целое число или число с плавающей точкой, реже указатель (то есть адрес в памяти). С предоставляет атомарные структуры, но даже в этом случае обычно нужен мьютекс для их защиты.

Однако случай простых операций со скаляром самый распространенный, и в этой ситуации можно отказаться от мьютексов, заменив их атомарными операциями, которые по существу захватывают неявный мьютекс при каждом использовании переменной.

Чтобы сообщить OpenMP, что вы хотите сделать с атомом, понадобится прагма:

```
#pragma omp atomic read
out = atom;

#pragma omp atomic write seq_cst
atom = out;

#pragma omp atomic update seq_cst
atom ++; //или atom--

#pragma omp atomic update
// или любая бинарная операция: atom *= x, atom /=x, ...
```

[1] Кстати, в стандарте С говорится, что у атомов С бесконечный период полураспада: «атомарные переменные не подвержены распаду» [C11 7.17.3(13), сноска].

```
atom -= x;

#pragma omp atomic capture seq_cst
// обновить-затем-прочитать
out = atom *= 2;
```

Фраза seq_cst необязательна, но рекомендуется (если компилятор ее поддерживает); я вернусь к ней чуть ниже.

Увидев прагму, компилятор должен сгенерировать команды, необходимые для того, чтобы на чтение атома в одной части кода не оказывала влияние запись в тот же атом в другой части.

В программе подсчета делителей все ресурсы, защищенные мьютексами, – скаляры, а значит, мьютексы нам в общем-то и не нужны. Использование атомов делает программу короче и понятнее (пример 12.6).

Пример 12.6 ❖ Распараллеленный поиск максимального элемента в массиве (openmp_getmax.c)

```
#include <omp.h>
#include <stdio.h>
#include <string.h> //memset

#include "openmp_getmax.c"

int main(){
    long int max = 1e7;
    int *factor_ct = malloc(sizeof(int)*max);

    factor_ct[0] = 0;
    factor_ct[1] = 1;
    for (long int i=2; i< max; i++)
      factor_ct[i] = 2;

    #pragma omp parallel for
    for (long int i=2; i<= max/2; i++)
      for (long int scale=2; scale*i < max; scale++) {
          #pragma omp atomic update
          factor_ct[scale*i]++;
      }

    int max_factors = get_max_factors(factor_ct, max);
    long int tally[max_factors+1];
    memset(tally, 0, sizeof(long int)*(max_factors+1));

    #pragma omp parallel for
    for (long int i=0; i< max; i++){
      #pragma omp atomic update
      tally[factor_ct[i]]++;
    }

    for (int i=0; i<=max_factors; i++)
      printf("%i\t%li\n", i, tally[i]);
}
```

Последовательная непротиворечивость

Хороший компилятор переупорядочивает последовательность операций, стремясь получить математически эквивалентную написанной вами, но работающую быстрее. Если переменная инициализируется в десятой строке, а впервые используется в двадцатой, то, возможно, будет быстрее совместить инициализацию и использование в двадцатой строке, чем выполнять два отдельных шага. Вот пример кода с двумя потоками, взятый из стандарта C11 §7.17.3(15) и сведенный к более понятному псевдокоду:

```
x = y = 0;

// Поток 1:
r1 = load(y);
store(x, r1);

// Поток 2:
r2 = load(x);
store(y, 42);
```

Когда мы читаем этот код на странице книги, кажется, что r2 не может принять значение 42, потому что 42 сохраняется в переменной y в строке, следующей за той, где производится присваивание r2. Если все команды из потока 1 исполняются до команд из потока 2, между двумя командами потока 2 или после последней команды, то r2 действительно не может стать равным 42. Но компилятор мог бы переставить местами две команды из потока 2, потому что в первой используются только переменные r2 и x, а во второй – только переменная y, и, значит, между ними нет зависимости, требующей, чтобы одна выполнялась раньше другой. Таким образом, допустима и такая последовательность:

```
x = y = 0;
store(y, 42); // поток 2
r1 = load(y); // поток 1
store(x, r1); // поток 1
r2 = load(x); // поток 2
```

Теперь все четыре переменные – y, x, r1 и r2 – равны 42.

В стандарте C описаны и еще более извращенные случаи, один из них даже снабжен комментарием «такое поведение нельзя назвать полезным, и реализация должна избегать его».

Вот к подобным ситуациям и относится фраза seq_cst: она сообщает компилятору, что атомарные операции в данном потоке должны производиться именно в том порядке, в котором написаны. Она была добавлена в спецификацию OpenMP 4.0, чтобы можно было воспользоваться преимуществами последовательно непротиворечивых атомов C, и ваш компилятор, возможно, еще не поддерживает ее. А пока рекомендуется внимательно следить за тонкими ошибками, которые могут возникнуть, если компилятор изменит порядок независимых строк, исполняемых в одном потоке.

Библиотека pthread

А теперь перепишем приведенный выше пример с использованием pthread. Элементы похожи: средства порождения и ожидания потоков и мьютексы. В pthread атомарных переменных нет, но они есть в самом C; см. ниже.

Существенная разница заключается в том, что функция pthread_create, создающая новый поток, принимает (помимо прочих аргументов) функцию типа void *fn(void *in), а поскольку эта функция принимает единственный указатель на void,

то приходится писать специфичную для данной функции структуру для передачи ей данных. Функция также возвращает указатель, хотя если мы все равно определяем специальный тип, то обычно проще включить в него выходные элементы наряду с входными, а не создавать специальную структуру для возврата данных.

Прежде чем приводить код целиком, я хотел бы отдельно рассмотреть несколько важнейших участков (это означает, что часть переменных в коде не определена):

```
tally_s thread_info[thread_ct];
for (int i=0; i< thread_ct; i++){
  thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                             .tally=tally, .max=max, .factor_ct=factor_ct,
                             .mutexes=mutexes, .mutex_ct =mutex_ct};
  pthread_create(&threads[i], NULL, add_tally, &thread_info[i]);
}
for (int t=0; t< thread_ct; t++)
  pthread_join(threads[t], NULL);
```

В первом цикле for создается фиксированное количество потоков (довольно трудно динамически получить от pthread число потоков, подходящее в конкретной ситуации). Сначала инициализируется структура, а затем мы вызываем pthread_create таким образом, чтобы в потоке исполнялась функция add_tally, которой передается специально подготовленная структура. В конце цикла будут работать thread_ct потоков.

Следующий цикл for – этап сбора результатов. Функция pthread_join блокирует выполнение до тех пор, пока указанный поток не завершит работу. Таким образом, мы окажемся в точке, следующей за циклом for, не раньше, чем все потоки завершатся, и в этот момент останется только один главный поток.

Библиотека pthread предоставляет мьютексы, которые ведут себя почти так же, как мьютексы OpenMP. В простых примерах ниже для перехода на мьютексы pthread достаточно изменить имена типов.

Ниже приведена программа подсчета делителей, переписанная с использованием библиотеки pthread. Помещение каждой подпрограммы в отдельный поток, определение передаваемых функциям структур и процедуры создания и ожидания завершения потоков требуют довольно много кода (при том, что я по-прежнему пользуюсь написанной для OpenMP функцией get_max).

Пример 12.7 ❖ Программа подсчета делителей, переписанная с использованием библиотеки pthread (pthread_factors.c)

```
#include <omp.h>      //get_max по-прежнему пользуется OpenMP
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>   //malloc
#include <string.h>   //memset

#include "openmp_getmax.c"

typedef struct {
  long int *tally;
  int *factor_ct;
```

```
  int max, thread_ct, this_thread, mutex_ct;
  pthread_mutex_t *mutexes;
} tally_s;

void *add_tally(void *vin){
  tally_s *in = vin;                                                    ❶
  for (long int i=in->this_thread; i < in->max; i += in->thread_ct){
     int factors = in->factor_ct[i];
     pthread_mutex_lock(&in->mutexes[factors% in->mutex_ct]);          ❷
     in->tally[factors]++;
     pthread_mutex_unlock(&in->mutexes[factors% in->mutex_ct]);
  }
  return NULL;
}

typedef struct {
  long int i, max, mutex_ct;
  int *factor_ct;
  pthread_mutex_t *mutexes ;
} one_factor_s;

void *mark_factors(void *vin){
  one_factor_s *in = vin;
  long int si = 2*in->i;
  for (long int scale=2; si < in->max; scale++, si=scale*in->i) {
     pthread_mutex_lock(&in->mutexes[si% in->mutex_ct]);
     in->factor_ct[si]++;
     pthread_mutex_unlock(&in->mutexes[si% in->mutex_ct]);
  }
  return NULL;
}

int main(){
  long int max = 1e7;
  int *factor_ct = malloc(sizeof(int)*max);

  int thread_ct = 4, mutex_ct = 128;
  pthread_t threads[thread_ct];
  pthread_mutex_t mutexes[mutex_ct];
  for (long int i=0; i< mutex_ct; i++)
     pthread_mutex_init(&mutexes[i], NULL);

  factor_ct[0] = 0;
  factor_ct[1] = 1;
  for (long int i=2; i< max; i++)
     factor_ct[i] = 2;

  one_factor_s x[thread_ct];
  for (long int i=2; i<= max/2; i+=thread_ct){
     // от лишних потоков вреда не будет
     for (int t=0; t < thread_ct && t+i <= max/2; t++){
        x[t] = (one_factor_s){.i=i+t, .max=max,
                             .factor_ct=factor_ct, .mutexes=mutexes,
                             .mutex_ct=mutex_ct};
        pthread_create(&threads[t], NULL, mark_factors, &x[t]);        ❸
```

```
    }
    for (int t=0; t< thread_ct; t++)
       pthread_join(threads[t], NULL);                          ❹
}
FILE *o=fopen("xpt", "w");
for (long int i=0; i < max; i ++){
   int factors = factor_ct[i];
   fprintf(o, "%i%li\n", factors, i);
}
fclose(o);

int max_factors = get_max(factor_ct, max);
long int tally[max_factors+1];
memset(tally, 0, sizeof(long int)*(max_factors+1));

tally_s thread_info[thread_ct];
for (int i=0; i< thread_ct; i++){
   thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                              .tally=tally, .max=max,
                              .factor_ct=factor_ct, .mutexes=mutexes,
                              .mutex_ct =mutex_ct};
   pthread_create(&threads[i], NULL, add_tally, &thread_info[i]);
}
for (int t=0; t< thread_ct; t++)
   pthread_join(threads[t], NULL);

for (int i=0; i<=max_factors; i++)
   printf("%i\t%li\n", i, tally[i]);
}
```

❶ Одноразовый псевдоним типа `tally_s` не только необходим для вызова `pthread_create`, но и существенно повышает безопасность. Мы, конечно, должны следить за правильностью входов и выходов системы pthread, но типы элементов структуры проверяются компилятором как в `main`, так и в этой функции-обертке. Если через неделю я решу в качестве `tally` использовать массив чисел типа `int`, но внесу изменения некорректно, компилятор предупредит меня.

❷ Мьютексы pthread и OpenMP очень похожи.

❸ Здесь создается поток. Мы подготавливаем массив указателей на данные потока, а затем по одному передаем их функции `pthread_create` вместе с функцией-оберткой и предназначенными для нее данными. Второй аргумент задает некоторые атрибуты потока, которые в этой главе вводного характера мы рассматривать не будем.

❹ Во втором цикле собираются результаты. Второй аргумент функции `pthread_join` – это адрес, по которому мы могли бы записать результат исполняемой потоком функции (`mark_factors`).



Фигурная скобка в конце цикла `for` закрывает область видимости, поэтому все объявленные в ней локальные переменные уничтожаются. Обычно мы не покидаем область видимости, пока не вернут управление все вызванные функции, но смысл `pthread_create` в том и состоит, чтобы функция `main` продолжала работать одновременно с потоком. Поэтому такой код не годится:

```
for (int i=0; i< 10; i++){
```

```
    tally_s thread_info = {...};
    pthread_create(&threads[i], NULL, add_tally, &thread_info);
}
```

потому что переменная `thread_info` уже будет уничтожена к тому моменту, как `add_tally` соберется ей воспользоваться. Перенос объявления вовне цикла:

```
tally_s thread_info;
for (int i=0; i< 10; i++){
    thread_info = (tally_s) {...};
    pthread_create(&threads[i], NULL, add_tally, &thread_info);
}
```

тоже работать не будет, потому что данные, хранящиеся в `thread_info`, будут затерты на второй итерации, хотя первая итерация еще не закончила их использовать. Поэтому мы создали массив данных для передачи функции и тем самым гарантировали, что данные потока не будут ни уничтожены, ни затерты во время запуска следующего потока.

Какую награду мы получаем от pthread за всю эту дополнительную работу? Больше возможностей. Например, `pthread_rwlock_t` – мьютекс, который блокирует поток, если какой-то другой поток изменяет защищаемый ресурс, но не препятствует одновременным операциям чтения. `pthread_cont_t` – семафор, который позволяет блокировать и разблокировать сразу несколько потоков после получения сигнала и может использоваться для реализации блокировок чтения-записи или мьютексов общего вида. Но чем больше гибкости, тем больше шансов сделать что-то не так. С помощью pthread очень легко написать тщательно оптимизированный многопоточный код, который будет работать быстрее, чем версия на OpenMP на сегодняшнем тестовом компьютере, но окажется абсолютно неработоспособным, на компьютере, выпущенном через год.

В спецификации OpenMP нет никаких упоминаний о pthread, а в спецификации POSIX не упоминается OpenMP, поэтому не существует сколько-нибудь официального документа, требующего, чтобы семантика *потока* в смысле OpenMP и в смысле POSIX совпадала. Однако авторы компилятора должны были изыскать какой-то способ реализовать OpenMP, POSIX (или Windows) и библиотеки многопоточности для С, и разрабатывать разные методы организации потоков для каждой спецификации оказалось бы слишком накладно. Кроме того, у процессора нет особых ядер для pthread и для OpenMP: у него имеется набор машинных команд для управления потоками, так что на компилятор возлагается задача свести все стандарты и спецификации к этому набору команд. Поэтому не будет таким уж грехом смешивать разные механизмы, например создать потоки с помощью прагмы OpenMP и использовать мьютексы pthread или атомы С для защиты ресурсов или начать с OpenMP, а затем в какой-то части программы применить функции pthread.

Атомы С

В стандарте С описаны заголовки `stdatomic.h` и `threads.h`, в которых объявлены функции и типы для работы с атомарными переменными и потоками. Ниже я приведу пример, в котором многопоточность организуется с помощью pthread, а для защиты переменных используются атомы С.

Существуют две причины, по которым я не использую потоки C. Во-первых, все примеры в этой книге протестированы, а на момент ее написания мне не удалось раздобыть комбинацию компилятора со стандартной библиотекой, в которой был бы реализован заголовок `threads.h`. И это понятно в свете второй причины: потоки C построены по образцу потоков C++, при проектировании которых за основу были взяты средства, имеющиеся в системе потоков как Windows, так и в POSIX, поэтому потоки C по существу представляют собой просто переименование существующих средств без добавления какой-то интригующей новой функциональности. А вот атомы C вносят-таки новую лепту.

Если имеется тип `my_type`, скалярный, структурный или любой другой, то сделать его атомарным можно следующим образом:

```
_Atomic(my_type) x
```

В одной из ближайших версий компилятора будет работать и конструкция

```
_Atomic my_type x
```

В этом варианте более понятно, что `_Atomic` – это на самом деле квалификатор типа, как `const`. Для определенных в стандарте целочисленных типов можно использовать сокращенную запись `atomic_int x`, `atomic_bool x` и т. д.

Простое объявление переменной как атомарной уже дает несколько преимуществ: `x++`, `--x`, `x *= y` и другие бинарные операции составного присваивания становятся потокобезопасными (C11 §6.5.2.4(2) и §6.5.16.2(3)). Эти операции, а также все потокобезопасные операции, рассматриваемые ниже, последовательно непротиворечивы (обсуждение этого понятия в контексте атомов OpenMP см. в разделе «Последовательная непротиворечивость» выше). На самом деле в спецификации OpenMP v4.0 §2.12.6 говорится, что атомы OpenMP и атомы C11 ведут себя одинаково. Однако для выполнения всех остальных операций необходимо пользоваться специальными функциями.

- Выполнить инициализацию путем вызова `atomic_init(&your_var, starting_val)`, которая устанавливает начальное значение «одновременно с инициализацией дополнительного состояния, которое может быть необходимо реализации для поддержки атомарного объекта» (C11 §7.17.2.2(2)). Эта функция не является потокобезопасной, поэтому вызывать ее следует до создания потоков или защищать мьютексом либо критической областью. Существует также макрос `ATOMIC_VAR_INIT`, который можно поместить в строке объявления для достижения того же эффекта, то есть варианты

  ```
  _Atomic int i = ATOMIC_VAR_INIT(12);
  //или
  _Atomic int x;
  atomic_init(&x, 12);
  ```

  эквивалентны.
- Использовать функцию `atomic_store(&your_var, x)`, чтобы потокобезопасным образом присвоить переменной `your_var` значение `x`.

- Использовать выражение x = atomic_load(&your_var), чтобы потокобезопасным образом прочитать значение переменной your_var и присвоить его переменной x.
- Использовать выражение x = atomic_exchange(&your_var, y), чтобы записать y в your_var и скопировать в x предыдущее значение your_var.
- Использовать выражение x = atomic_fetch_add(&your_var, 7), чтобы прибавить семь к переменной your_var и записать в x значение, которое эта переменная имела до сложения; функция atomic_fetch_sub аналогично производит вычитание, но не существует функций atomic_fetch_mul и atomic_fetch_div.

Об атомарных переменных можно говорить еще долго, в том числе и потому, что комитет по стандартизации С надеется, что в будущих реализациях библиотек многопоточности атомарные переменные будут использоваться для создания мьютексов и других подобных конструкций в рамках стандартного С. Поскольку я не предполагаю, что вы станете реализовывать мьютексы самостоятельно, то и не буду останавливаться на соответствующих средствах (в частности, на функциях atomic_compare_exchange_weak и _strong, предназначенных для реализации операции «сравнить и обменять»).

Ниже приведен наш сквозной пример, переписанный с использованием атомарных переменных. Я использую для организации многопоточности библиотеку pthread, поэтому программа достаточно длинная, но все словеса, связанные с мьютексами, устранены.

Пример 12.8 ❖ Подсчет делителей с использованием атомарных переменных (c_factors.c)

```
#include <pthread.h>
#include <stdatomic.h>
#include <stdlib.h> //malloc
#include <string.h> //memset
#include <stdio.h>

int get_max_factors(_Atomic(int) *factor_ct, long int max){
   // в одном потоке, чтобы избежать словоблудия
   int global_max=0;
   for (long int i=0; i< max; i++){
      if (factor_ct[i] > global_max)
      global_max = factor_ct[i];
   }
   return global_max;
}

typedef struct {
   _Atomic(long int) *tally;
   _Atomic(int) *factor_ct;
   int max, thread_ct, this_thread;
} tally_s;

void *add_tally(void *vin){
   tally_s *in = vin;
```

```
    for (long int i=in->this_thread; i < in->max; i += in->thread_ct){
        int factors = in->factor_ct[i];
        in->tally[factors]++;                                          ❶
    }
    return NULL;
}

typedef struct {
    long int i, max;
    _Atomic(int) *factor_ct;
} one_factor_s;

void *mark_factors(void *vin){
    one_factor_s *in = vin;
    long int si = 2*in->i;
    for (long int scale=2; si < in->max; scale++, si=scale*in->i) {
        in->factor_ct[si]++;
    }
    return NULL;
}

int main(){
    long int max = 1e4;
    _Atomic(int) *factor_ct = malloc(sizeof(_Atomic(int))*max);       ❷

    int thread_ct = 4;
    pthread_t threads[thread_ct];

    atomic_init(factor_ct, 0);
    atomic_init(factor_ct+1, 1);
    for (long int i=2; i< max; i++)
        atomic_init(factor_ct+i, 2);

    one_factor_s x[thread_ct];
    for (long int i=2; i<= max/2; i+=thread_ct){
        for (int t=0; t < thread_ct && t+i <= max/2; t++){
            x[t] = (one_factor_s){.i=i+t, .max=max,
                                  .factor_ct=factor_ct};
            pthread_create(&threads[t], NULL, mark_factors, x+t);
        }
        for (int t=0; t< thread_ct && t+i <=max/2; t++)
            pthread_join(threads[t], NULL);
    }

    int max_factors = get_max_factors(factor_ct, max);
    _Atomic(long int) tally[max_factors];
    memset(tally, 0, sizeof(long int)*max_factors);

    tally_s thread_info[thread_ct];
    for (int i=0; i< thread_ct; i++){
        thread_info[i] = (tally_s){.this_thread=i, .thread_ct=thread_ct,
                                   .tally=tally, .max=max,
                                   .factor_ct=factor_ct};
        pthread_create(&threads[i], NULL, add_tally, thread_info+i);
```

```
  }
  for (int t=0; t< thread_ct; t++)
    pthread_join(threads[t], NULL);
  for (int i=0; i<max_factors; i++)
    printf("%i\t%li\n", i, tally[i]);
}
```

❶ Раньше для защиты этой строки мы использовали мьютекс или прагму `#pragma omp atomic`. Но поскольку элементы массивы `tally` объявлены атомарными, гарантируется, что простые арифметические операции будут потокобезопасными и так.

❷ Ключевое слово `_Atomic` – квалификатор типа, как и `const`. Но, в отличие от `const`, типы `atomic int` и просто `int` могут не совпадать по размеру (C11 §6.2.5(27)).

Атомарные структуры

Структуры могут быть атомарными. Однако «доступ к члену атомарной структуры или объединения приводит к неопределенному поведению» (C11 §6.5.2.3(5)). Поэтому для работы с ними необходима определенная дисциплина:

- скопировать разделяемую атомарную структуру в неатомарную локальную структуру того же базового типа: `struct_t private_struct = atomic_load (&shared_struct);`
- выполнить необходимые операции с локальной копией;
- скопировать модифицированную локальную копию обратно в атомарную структуру: `atomic_store(&shared_struct, private_struct)`.

Если одну структуру могут модифицировать два потока, то нет никакой гарантии, что между чтением на первом шаге и записью на третьем структура не будет изменена. Поэтому следует позаботиться о том, чтобы в каждый момент времени запись мог производить только один поток – в силу логики программы или с помощью мьютексов. Однако для чтения структуры мьютекс больше не нужен.

Рассмотрим специализированную программу поиска простых чисел. Примененный выше метод «просеивания» (вариант решета Эратосфена) в моих тестах искал простые числа гораздо быстрее, но эта версия демонстрирует элегантное использование атомарной структуры.

Я хочу проверить, что число-кандидат не делится нацело ни на какое меньшее число. Но если число не делится на 3 и не делится на 5, то оно заведомо не делится на 15, поэтому нужно лишь проверять, что число не делится на меньшие простые числа. Кроме того, не имеет смысла проверять множители, большие половины числа-кандидата, потому что максимально возможный множитель удовлетворяет соотношению 2 * множитель = кандидат. Таким образом, можно написать псевдокод:

```
for (candidate in 2 to a million){
    is_prime = true
    for (test in (простые числа, меньшие candidate/2))
      if ((candidates/test) делится без остатка)
        is_prime = false
}
```

Остается единственная проблема – поддерживать список простых чисел, меньших candidate/2). Нам нужен список переменного размера, то есть придется вызывать realloc. Я собираюсь использовать простой массив без маркера конца, поэтому нужно будет где-то хранить его длину. Мы имеем идеальную ситуацию для применения атомарной структуры, потому что массив и его длина должны быть синхронизированы.

В приведенной ниже программе prime_list – структура, разделяемая всеми потоками. Как видите, ее адрес несколько раз передается функциям в виде аргумента, но во всех остальных случаях он используется лишь в atomic_init, atomic_store и atomic_load. Структура модифицируется только в функции add_a_prime, и делается это, как описано выше: сначала копирование в локальную структуру, а затем манипуляции с этой локальной структурой. Эти действия защищены мьютексом, потому что одновременное обращение к realloc из двух потоков привело бы к катастрофе.

В функции test_a_number есть еще одно тонкое место: она ждет, пока в списке prime_list не окажутся все простые числа, не превышающие candidate/2, и только потом приступает к работе, чтобы не пропустить какой-нибудь множитель. Это работает благодаря свойствам простых чисел; можно доказать, что программа не окажется в ситуации *взаимоблокировки*, когда каждый поток ждет завершения другого, чтобы продолжить работу. В остальном алгоритм точно следует написанному выше псевдокоду. Отметим, что в этой части кода нет мьютексов, потому что atomic_load используется только для чтения структуры.

Пример 12.9 ❖ Использование атомарной структуры для поиска простых чисел (c_primes.c)

```
#include <stdio.h>
#include <stdatomic.h>
#include <stdlib.h> //malloc
#include <stdbool.h>
#include <pthread.h>

typedef struct {                                  ❶
    long int *plist;
    long int length;
    long int max;
} prime_s;

int add_a_prime(_Atomic (prime_s) *pin, long int new_prime){
    prime_s p = atomic_load(pin);                 ❷
    p.length++;
    p.plist = realloc(p.plist, sizeof(long int) * p.length);
    if (!p.plist) return 1;
    p.plist[p.length-1] = new_prime;
    if (new_prime > p.max) p.max = new_prime;
    atomic_store(pin, p);
    return 0;
}

typedef struct{
    long int i;
```

```
    _Atomic (prime_s) *prime_list;
    pthread_mutex_t *mutex;
} test_s;

void* test_a_number(void *vin){
    test_s *in = vin;
    long int i = in->i;
    prime_s pview;
    do {
       pview = atomic_load(in->prime_list);
    } while (pview.max*2 < i);

    bool is_prime = true;
    for (int j=0; j < pview.length; j++)
       if (!(i% pview.plist[j])){
           is_prime = false;
           break;
       }

    if (is_prime){
       pthread_mutex_lock(in->mutex);
       int retval = add_a_prime(in->prime_list, i);
       if (retval) {printf("Слишком много простых чисел.\n"); exit(0);}
       pthread_mutex_unlock(in->mutex);
    }
    return NULL;
}

int main(){
    prime_s inits = {.plist=NULL, .length=0, .max=0};
    _Atomic (prime_s) prime_list = ATOMIC_VAR_INIT(inits);

    pthread_mutex_t m;
    pthread_mutex_init(&m, NULL);

    int thread_ct = 3;
    test_s ts[thread_ct];
    pthread_t threads[thread_ct];

    add_a_prime(&prime_list, 2);
    long int max = 1e6;
    for (long int i=3; i< max; i+=thread_ct){
       for (int t=0; t < thread_ct && t+i < max; t++){
           ts[t] = (test_s) {.i = i+t, .prime_list=&prime_list, .mutex=&m};
           pthread_create(threads+t, NULL, test_a_number, ts+t);
       }
       for (int t=0; t< thread_ct && t+i <max; t++)
           pthread_join(threads[t], NULL);
    }

    prime_s pview = atomic_load(&prime_list);
    for (int j=0; j < pview.length; j++)
       printf("%li\n", pview.plist[j]);
}
```

❶ Сам список и его длина должны быть синхронизированы на всем протяжении перераспределения памяти, поэтому мы помещаем их в структуру и объявляем только атомарные экземпляры этой структуры.

❷ В этой функции применяется регламентированная процедура: загрузка атомарной структуры в локальную неатомарную копию, модификация копии и обратное сохранение копии в атомарной структуре. Эта процедура не является потокобезопасной, поэтому в каждый момент времени выполнять ее может только один поток.

❸ Поскольку функция `add_a_prime` не потокобезопасная, защищаем обращения к ней мьютексом.

В этой главе мы рассмотрели ряд способов параллельного выполнения кода. Применение OpenMP позволяет очень просто – с помощью одной аннотации – создавать потоки и ожидать их завершения. Сложности возникают с отслеживанием переменных: все переменные, встречающиеся в распараллеленном участке программы, должны быть классифицированы и обрабатываться соответственно.

Проще всего обстоит дело с переменными, которые только читаются; за ними следуют переменные, которые создаются и уничтожаются в пределах одного потока и, стало быть, не взаимодействуют с другими потоками. Отсюда вытекает, что функции следует писать так, чтобы они не модифицировали входных аргументов (то есть все указатели должны быть помечены квалификатором `const`) и не имели других побочных эффектов. Такие функции можно выполнять параллельно без опасений. В каком-то смысле такие функции существуют вне времени и окружения: так, если `sum` – функция суммирования, то вызов `sum(2, 2)` всегда возвращает 4, как бы часто он ни выполнялся и что бы ни происходило в другом месте. Существуют так называемые *чисто функциональные* языки, которые всячески поощряют пользователя работать только с подобными функциями.

Переменные состояния изменяются в процессе выполнения функции. Если функция содержит переменные состояния, то она теряет не запачканную временем чистоту. Если выполнить функцию, возвращающую баланс счета, сегодня и завтра, то полученная величина может различаться. Философия приверженцев чисто функционального программирования сводится к простому правилу: держаться подальше от переменных состояния. Но они возникают неизбежно, потому что наши программы описывают мир, полный состояний. Читая работы, посвященные функциональному программированию, забавно следить, насколько далеко автор сможет зайти, не упомянув о состоянии. Например, Абельсон [abelson-1996] прошел примерно треть пути (до стр. 217) и только потом признал, что мир изобилует состояниями: балансы банковских счетов, генераторы псевдослучайных чисел, электрические цепи.

Большая часть этой главы была посвящена тому, как обращаться с переменными состояния в распараллеленном окружении, после того как время распалось. В нашем распоряжении есть несколько инструментов снова склеить время, в том числе атомарные операции, мьютексы и критические области. Все они позволяют сериализовать операции изменения состояния. Но поскольку требуется время на их реализацию, верификацию и отладку, то проще всего избегать переменных состояния, а к функциям, зависящим от времени и окружения, прибегать только в крайнем случае.



Библиотеки

И если бы я действительно хотел чему-то научиться, то слушал бы новые записи. Так поступаю я. И вы. И мы.

— The Hives «Untutored Youth»

В этой главе мы рассмотрим несколько библиотек, которые могут облегчить жизнь программиста.

Мне кажется, что с течением времени написанные на C библиотеки стали менее педантичными. Еще десять лет назад типичная библиотека предлагала минимальный набор средств, и ожидалось, что удобные для программирования надстройки над ними вы напишете сами. Предполагалось, что вы сами будете выделять память, потому что недостойно библиотеки цапать память без спросу. Напротив, библиотеки, представленные в этой главе, предоставляют «простой» интерфейс, как, например, функции типа `curl_easy_...` в cURL или единственная функция Sqlite, которая выполняет все неприглядные шаги транзакции базы данных. Если функции понадобится для работы промежуточная память, она выделит ее. Работать с такими функциями – одно удовольствие.

Я начну с относительно стандартных библиотек общего характера, а потом перейду к своим любимым библиотекам более узкой направленности: SQLite, GNU Scientific Library, libxml2 и libcURL. Я не знаю, для чего используете язык C вы, но эти библиотеки представляют собой удобные и надежные средства решения широкого круга задач.

GLib

Поскольку стандартная библиотека отнюдь не всеобъемлюща, естественно, что со временем появились библиотеки, заполняющие пробелы. В библиотеке GLib реализовано столько базовых вычислительных средств, что она вполне могла бы сдать за вас экзамен за первый курс обучения информатике. Кроме того, она перенесена практически на все платформы (включая даже издания Windows, не поддерживающие POSIX) и достаточно стабильна, так что на нее можно положиться.

Я не стану специально приводить примеры кода с использованием GLib, поскольку они уже встречались ранее, а именно:

- краткое введение в связанные списки (пример 2.2);
- тестовая оснастка в разделе «Автономное тестирование»;
- средства работы с Unicode в разделе «Unicode»;

- хэши в разделе «Обобщенные структуры»;
- считывание текстового файла в память и использование совместимых с Perl регулярных выражений в разделе «Подсчет ссылок».

А ниже, в разделе «Использование mmap при работе с очень большими наборами данных», я еще упомяну имеющиеся в GLib средства для обертывания системного вызова `mmap` в POSIX и в Windows.

Но и это не все. При написании оконной программы, в которой используется мышь, вам понадобится цикл обработки событий для перехвата и диспетчеризации событий мыши и клавиатуры – в GLib он есть. Включены также средства для работы с файлами, которые правильно ведут себя в системах, совместимых со стандартом POSIX и прочих (читай: в Windows). Имеются также простой анализатор конфигурационных файлов и облегченный лексический анализатор для более сложных задач. И так далее.

Стандарт POSIX

Стандарт POSIX добавил несколько полезных функций в стандартную библиотеку C. Учитывая, насколько широко распространен POSIX, знания об этих функциях будут нелишними. Ниже я расскажу об использовании двух особенно полезных средств: разбор регулярных выражений и проецирование файла на память.

Разбор регулярных выражений

Регулярные выражения предлагают нотацию, позволяющую записывать образцы для сравнения с текстом, например «число, за которым следует одна или более букв» или «строка должна содержать число, запятую, пробел, число и больше ничего». Этим словесным описаниям соответствуют такие регулярные выражения: `[0-9]\+[[:alpha:]]\+` и `^[0-9]\+, [0-9]\+\$`. Стандарт POSIX определяет набор C-функций для разбора регулярных выражений, записанных с соблюдением определенных грамматических правил. Эти функции включены в сотни инструментов. Не будет преувеличением сказать, что я пользуюсь ими ежедневно – либо в виде совместимых с POSIX утилит (`sed`, `awk`, `grep`), либо непосредственно в своих программах для разбора текста. Например, они позволяют найти имя человека в файле или, имея диапазон дат вида `"04Apr2009-12Jun2010"`, выделить из этой строки шесть полей. Или найти маркеры глав в беллетризованном трактате о китах.



Для разбиения строки на лексемы, разделенные одним каким-то символом, лучше использовать функцию `strtok`. См. раздел «Песнь о strtok» на стр. 199.

Однако я решил не включать в эту книгу руководство по регулярным выражениям. Поиск по запросу «*regular expression tutorial*» дал 12 900 ссылок. На машине с Linux команда `man 7 regex` напечатает краткую справку, а если у вас установлен Perl, то команда `man perlre` расскажет о совместимых с Perl регулярных выражениях (PCRE). В работе [Friedl 2002] вы найдете отличное освещение темы с приличествующими книге подробностями. Здесь же я покажу только, как регулярные выражения работают в библиотеке C, совместимой с POSIX.

Существует три основных типа регулярных выражений.

- Базовые регулярные выражения (BRE) были представлены в первоначальном варианте. В них было всего несколько специальных символов, например * означала 0 или более повторений предшествующего атома, то есть выражение [0-9]* представляло необязательное целое число. Для использования дополнительных возможностей нужно было употреблять знак обратной косой черты, предваряющий специальный символ; так, *одна или несколько цифр* обозначались конструкцией \+, поэтому целое число со знаком плюс можно было представить выражением +[0-9]\+.
- Расширенные регулярные выражения (ERE) – второй вариант. Основное отличие – употребление специальных символов без обратной косой черты и экранирование их в обычном тексте с помощью обратной косой черты. Теперь регулярное выражение для целого числа со знаком плюс стало записываться в виде \+[0-9]+.
- В Perl регулярные выражения встроены в сам язык, и авторы внесли ряд существенных добавлений в их грамматику, в том числе возможность заглядывания вперед и оглядывания назад, нежадные кванторы, удовлетворяющиеся минимально возможным совпадением, и внутренние комментарии.

Первые два типа регулярных выражений реализованы в виде небольшого набора функций, определенных в стандарте POSIX. Скорее всего, они есть в вашей стандартной библиотеке. Совместимые с Perl регулярные выражения находятся в библиотеке libpcre, которую можно скачать из Сети или установить с помощью менеджера пакетов. Подробное описание функций из этой библиотеки дает команда man pcreapi. Glib предоставляет удобную высокоуровневую надстройку над libpcre; как она используется, см. в примере 11.18.

Поскольку регулярные выражения – неотъемлемая часть POSIX, программа, приведенная в примере 13.2, будет компилироваться на платформах Linux и Mac без всяких флагов компилятора, помимо необходимых:

```
CFLAGS="-g -Wall -O3 --std=gnu11" make regex
```

Оба интерфейса, POSIX и PCRE, используются одинаково, процедура состоит из четырех шагов:

- откомпилировать регулярное выражение с помощью функции regcomp или pcre_compile;
- сопоставить строку с откомпилированным регулярным выражением с помощью функции regexec или pcre_exec;
- если в регулярном выражении были запоминаемые группы, то выделить соответствующие им части строки можно, воспользовавшись смещениями, которые возвращает функция regexec или pcre_exec;
- освободить память, отведенную для откомпилированного выражения.

Для выполнения первых двух и последнего шага достаточно одной строки кода, так что если вас интересует только, соответствует ли строка регулярному выражению, то все просто. Не буду вдаваться в детальное обсуждение флагов и ис-

пользования функций regcomp, regexec и regfree, потому что раздел относящегося к ним стандарта POSIX воспроизведен в страницах руководства в Linux и BSD (наберите команду man regexec), а также на *очень* многих сайтах.

Если требуется выделить подстроки, то ситуация немного усложняется. Круглые скобки внутри регулярного выражения означают, что требуется найти соответствие частичному образцу в этих скобках (даже если ему соответствует только пустая строка). Таким образом, образец "(.*)o" в грамматике ERE сопоставляется со строкой "hello" и в качестве побочного эффекта сохраняет самое длинное совпадение с .*, каковым является hell. Третий аргумент функции regexec – это количество заключенных в скобки подвыражений, в примере ниже я назвал его matchcount. Четвертый аргумент regexec – массив из matchcount+1 элементов типа regmatch_t. Тип regmatch_t состоит из двух полей: rm_so содержит начало совпавшей подстроки, а rm_eo – ее конец. В нулевом элементе массива хранятся начало и конец совпадения со всем регулярным выражением (представьте, что все оно заключено в скобки), а в последующих элементах – начало и конец подстроки, совпавшей с каждым подвыражением. Подвыражения нумеруются в порядке появления открывающих скобок.

Предвосхищая грядущее, в примере 13.1 показан заголовок, в котором объявлены две служебные функции, реализованные в конце этого раздела. Функция regex_match и сопутствующие ей макрос и структура допускают именованные и необязательные аргументы (см. раздел «Необязательные и именованные аргументы» на стр. 231). Эта функция принимает строку и регулярное выражение и возвращает массив подстрок.

Пример 13.1 ❖ Заголовок с объявлениями служебных функций для работы с регулярными выражениями (regex_fns.h)

```
typedef struct {
  const char *string;
  const char *regex;
  char ***substrings;
  _Bool use_case;
} regex_fn_s;

#define regex_match(...) regex_match_base((regex_fn_s){__VA_ARGS__})

int regex_match_base(regex_fn_s in);
char * search_and_replace(char const *base, char const *search,
                          char const *replace);
```

Отдельная функция для поиска и замены необходима, потому что в POSIX ничего подобного нет. Если длины заменяемой и заменяющей строк различаются, то необходимо перераспределить память для исходной строки. Но у нас уже есть средства для разбиения строки на подстроки, поэтому search_and_replace использует подстроки, соответствующие заключенным в скобки подвыражениям, чтобы разбить строку на части, а затем строит новую строку, вставляя в нужные места заменяющие подстроки.

Функция возвращает NULL, если совпадение не найдено, поэтому можно следующим образом выполнить глобальный поиск и замену:

```
char *s2;
while((s2 = search_and_replace(long_string, pattern))){
  char *tmp = long_string;
  long_string = s2;
  free(tmp);
}
```

Этот код не очень эффективен: regex_match каждый раз заново компилирует регулярное выражение, а глобальный поиск и замена был бы эффективнее, если бы учитывался тот факт, что в части строки, предшествующей result[1].rm_eo, повторно искать не надо. В данном случае мы можем использовать C как язык разработки прототипа для C: сначала пишем простой вариант, а если профилировщик показывает, что этот вариант неэффективен, то заменяем его более эффективным.

Ниже приведен сам код. Особо интересные строки помечены, а после кода приведены дополнительные замечания. В конце имеется тестовая функция, демонстрирующая способы использования.

Пример 13.2 ❖ Служебные функции для работы с регулярными выражениями (regex.c)

```
#define _GNU_SOURCE //чтобы stdio.h включал asprintf
#include "stopif.h"
#include <regex.h>
#include "regex_fns.h"
#include <string.h> //strlen
#include <stdlib.h> //malloc, memcpy

static int count_parens(const char *string){                  ❶
   int out = 0;
   int last_was_backslash = 0;
   for(const char *step=string; *step !='\0'; step++){
      if (*step == '\\' && !last_was_backslash){
          last_was_backslash = 1;
          continue;
      }
      if (*step == ')' && !last_was_backslash)
          out++;
      last_was_backslash = 0;
   }
   return out;
}

int regex_match_base(regex_fn_s in){
   Stopif(!in.string, return -1, "строка равна NULL");
   Stopif(!in.regex, return -2, "регулярное выражение равно NULL");

   regex_t re;
   int matchcount = 0;
   if (in.substrings) matchcount = count_parens(in.regex);
      regmatch_t result[matchcount+1];
```

```
    int compiled_ok = !regcomp(&re, in.regex, REG_EXTENDED                    ❷
                                + (in.use_case ? 0 : REG_ICASE)
                                + (in.substrings ? 0 : REG_NOSUB) );
    Stopif(!compiled_ok, return -3,
      "Ошибка при компиляции регулярного выражения: \"%s\"", in.regex);

    int found = !regexec(&re, in.string, matchcount+1, result, 0);            ❸
    if (!found) return 0;
    if (in.substrings){
      *in.substrings = malloc(sizeof(char*) * matchcount);
      char **substrings = *in.substrings;
      // совпадение с индексом 0 соответствует всей строке, игнорируем
      for (int i=0; i < matchcount; i++){
          if (result[i+1].rm_eo > 0){ // особенность GNU: совпадение с пустой
                                      // подстрокой помечается как -1
              int length_of_match = result[i+1].rm_eo - result[i+1].rm_so;
              substrings[i] = malloc(strlen(in.string)+1);
              memcpy(substrings[i], in.string + result[i+1].rm_so,
                      length_of_match);
              substrings[i][length_of_match] = '\0';
          } else { // совпадение с пустой подстрокой
              substrings[i] = malloc(1);
              substrings[i][0] = '\0';
          }
      }
      in.string += result[0].rm_eo; // конец совпадения со всем выражением
    }
    regfree(&re);                                                             ❹
    return matchcount;
}

char * search_and_replace(char const *base, char const *search,
                          char const *replace){
    char *regex, *out;
    asprintf(&regex, "(.*)(%s)(.*)", search);                                 ❺
    char **substrings;
    int match_ct = regex_match(base, regex, &substrings);
    if(match_ct < 3) return NULL;
    asprintf(&out, "%s%s%s", substrings[0], replace, substrings[2]);
    for (int i=0; i< match_ct; i++)
      free(substrings[i]);
    free(substrings);
    return out;
}

#ifdef test_regexes
int main(){
    char **substrings;
    int match_ct = regex_match("Hedonism by the alps, savory foods "
                               "at every meal.",
            "([He]*)do.*a(.*)s, (.*)or.* ([em]*)al", &substrings);
    printf("%i matches:\n", match_ct);
    for (int i=0; i< match_ct; i++){
```

```
        printf("[%s] ", substrings[i]);
        free(substrings[i]);
    }
    free(substrings);
    printf("\n\n");
    match_ct = regex_match("", "([[:alpha:]]+) ([[:alpha:]]+)", &substrings);
    Stopif(match_ct != 0, return 1, "Error: matched a blank");
    printf("Without the L, Plants are:%s",
          search_and_replace("Plants\n", "l", ""));
}
#endif
```

❶ Мы должны передать regexec массив для хранения позиций совпавших подстрок и его длину, то есть предполагается, что мы знаем, сколько будет подстрок. Эта функция принимает регулярное выражение и подсчитывает, сколько в нем открывающих круглых скобок, не экранированных знаком обратной косой черты.

❷ Здесь регулярное выражение компилируется в значение типа regex_t. При повторном использовании эта функция неэффективна, так как выражение всякий раз компилируется заново. Оставляю читателю в качестве упражнения реализовать кэширование уже откомпилированных регулярных выражений.

❸ Здесь используется regexec. Если интересует только, есть совпадение или нет, то можно передать в качестве списка совпадений NULL, а в качестве его длины 0.

❹ Не забывайте освобождать внутреннюю память, отведенную для regex_t.

❺ Принцип работы поиска и замены – разбить входную строку на отрезки (все до совпадения)(совпадение)(все после совпадения). Это регулярное выражение решает поставленную задачу.

Использование mmap для очень больших наборов данных

Я уже упоминал о трех типах памяти (статическая, динамическая и автоматическая). Но есть еще и четвертый: дисковая. Системный вызов mmap позволяет спроецировать хранящийся на диске файл в область памяти.

Именно так часто работают разделяемые библиотеки: система находит файл libwhatever.so и назначает отрезку этого файла, представляющему нужную функцию, адрес в памяти. Всё – функция загружена в память.

Или другой пример использования – чтобы данные стали доступны нескольким процессам, эти процессы должны спроецировать в память один и тот же файл.

Этим механизмом можно воспользоваться и для сохранения структур данных в памяти. Спроецируем файл в память, с помощью функции memmove скопируем находящуюся в памяти структуру данных в спроецированную область, и готово – структура сохранена до следующего раза. Проблемы возникают, когда структура данных содержит указатель на другие данные; преобразование последовательности указателей в нечто, допускающее сохранение, называется *сериализацией*, я эту задачу рассматривать не буду.

И, разумеется, проецирование позволяет работать с большими наборами данных, не помещающимися в память. Размер спроецированного в память массива ограничен размером диска, а не оперативной памяти.

В примере 13.3 приведена демонстрационная программа. Большую часть работы выполняет функция load_mmap. Если она используется в качестве malloc, то должна создать файл и «растянуть» его до нужного размера; если мы открываем уже существующий файл, то достаточно открыть и вызвать mmap.

Пример 13.3 ❖ Файл на диске можно прозрачно спроецировать в память (mmap.c)

```
#include <stdio.h>
#include <unistd.h> //lseek, write, close
#include <stdlib.h> //exit
#include <fcntl.h> //open
#include <sys/mman.h>
#include "stopif.h"

#define Mapmalloc(number, type, filename, fd) \                          ❶
       load_mmap((filename), &(fd), (number)*sizeof(type), 'y')
#define Mapload(number, type, filename, fd) \
       load_mmap((filename), &(fd), (number)*sizeof(type), 'n')
#define Mapfree(number, type, fd, pointer) \
       releasemmap((pointer), (number)*sizeof(type), (fd))

void *load_mmap(char const *filename, int *fd, size_t size, char make_room){ ❷
   *fd=open(filename,
          make_room=='y' ? O_RDWR | O_CREAT | O_TRUNC : O_RDWR,
          (mode_t)0600);
   Stopif(*fd==-1, return NULL, "Ошибка при открытии файла");

   if (make_room=='y'){ //Растянуть файл до размера спроецированного массива
      int result=lseek(*fd, size-1, SEEK_SET);
      Stopif(result==-1, close(*fd); return NULL,
            "Ошибка при растягивании файла с помощью lseek");
      result=write(*fd, "", 1);
      Stopif(result!=1, close(*fd); return NULL,
            "Ошибка при записи последнего байта файла");
   }

   void *map=mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED, *fd, 0);
   Stopif(map==MAP_FAILED, return NULL, "Ошибка при проецировании файла");
   return map;
}

int releasemmap(void *map, size_t size, int fd){                          ❸
   Stopif(munmap(map, size) == -1, return -1,
         "Ошибка при отмене проецирования файла");
   close(fd);
   return 0;
}

int main(int argc, char *argv[]) {
   int fd;
   long int N=1e5+6;
   int *map = Mapmalloc(N, int, "mmapped.bin", fd);

   for (long int i = 0; i <N; ++i) map[i] = i;                            ❹
```

```
    Mapfree(N, int, fd, map);

    // Открыть заново и выполнить подсчеты.
    int *readme = Mapload(N, int, "mmapped.bin", fd);

    long long int oddsum=0;
    for (long int i = 0; i <N; ++i) if (readme[i]%2) oddsum += i;
    printf("Сумма нечетных чисел до%li:%lli\n", N, oddsum);

    Mapfree(N, int, fd, readme);
}
```

❶ Я обернул следующие далее функции макросами, чтобы не приходилось всякий раз печатать `sizeof` и не нужно было помнить о том, как вызывать `load_mmap` для выделения память, а как для загрузки.

❷ Макросы скрывают тот факт, что функцию можно вызывать двумя способами. Если нужно только открыть существующие данные, то мы открываем файл, вызываем `mmap` и проверяем коды возврата. Если же функция используется для выделения памяти, то файл необходимо растянуть до нужной длины.

❸ Для отмены проецирования нужно вызвать функцию `munmap`, в чем-то похожую на функцию `free`, парную `malloc`, и закрыть файл. Данные останутся на диске, так что на следующий день можно прийти, открыть тот же файл и продолжить с прерванного места. Чтобы вообще удалить файл, вызовите `unlink("filename")`.

❹ Награда: невозможно сказать, что массив `map` находится на диске, а не в обычной памяти.

Несколько деталей: функция `mmap` определена в стандарте POSIX, поэтому доступна всюду, кроме Windows и некоторых встраиваемых систем. В Windows имеются идентичные механизмы, но с другими именами и флагами; см. документацию по функциям `CreateFileMapping` и `MapViewOfFile`. Библиотека GLib обертывает `mmap` и функции Windows с помощью конструкций вида `if POSIX ... else if Windows ...` и называет все это `g_mapped_file_new`.

Библиотека GNU Scientific Library

Если кто-нибудь задаст вам вопрос, начинающийся словами «Я пытаюсь реализовать что-то из описанного в книге *Numerical Recipes in C...* [Press 1992]», то почти наверняка в ответ нужно посоветовать скачать библиотеку GNU Scientific Library (GSL), потому что там все уже сделано [Gough 2003].

Одни методы интегрирования функций лучше, другие хуже, и, как сказано в разделе «Нерекомендуемый тип float» на стр. 164, некоторые кажущиеся вполне разумными численные алгоритмы дают результаты настолько неточные, что правильными их нельзя назвать даже с натяжкой. Поэтому в области численного анализа применение существующих библиотек всегда окупается.

И уж как минимум GSL предлагает генератор случайных чисел (совместимый со стандартом C генератор на разных машинах может быть различен, что

не позволяет использовать его для получения воспроизводимых результатов) и структуры вектора и матрицы, которыми легко манипулировать. Даже если вы не занимаетесь денно и нощно численными расчетами, стандартные алгоритмы линейной алгебры, средства поиска минимумов функций, простейшие статистические показатели (среднее и дисперсия) и генераторы перестановок могут оказаться полезны.

А если вы знаете, что такое собственный вектор, бесселева функция или быстрое преобразование Фурье, то найдете функции для их вычисления.

Я приведу пример использования GSL в примере 13.4, хотя имена, начинающиеся префиксом `gsl_`, встречаются в нем всего один-два раза. GSL – отличный пример библиотеки старой школы, которая предоставляет минимальный набор необходимых инструментов, предполагая, что остальное вы напишете сами. Например, в руководстве по GSL приводится целая страница стереотипного кода, который нужно написать в дополнение к предлагаемым функциям оптимизации, чтобы достичь желаемого эффекта. Невозможно избавиться от ощущения, что библиотека должна бы делать это сама, поэтому я написал набор функций, обертывающих некоторые части GSL, и в результате получилась библиотека *Apophenia*, предназначенная для моделирования. Например, функция `apop_data` связывает исходные матрицы и векторы GSL с именами строк и столбцов и массивом текстовых данных, что делает базовые структуры для численных расчетов ближе к тому, как выглядят реальные данные. Соглашения о вызове библиотечных функций следуют современным принципам, изложенным в главе 10.

Оптимизатор устроен, как описано в разделе «Указатель на void и структура, на которую он указывает» на стр. 239: функция принимает произвольную функцию и использует ее как черный ящик. Оптимизатор подает данные на вход полученной функции и использует возвращенное значение для вычисления следующего входа, который должен дать большее значение. При достаточно изощренном алгоритме поиска последовательность входных данных сойдется к значению, на котором функция обращается в максимум. Таким образом, применение оптимизатора сводится к задаче написания оптимизируемой функции в надлежащей форме и передаче ее вместе с настройками оптимизатору.

Для примера предположим, что дан список точек x_1, x_2, x_3, ... в некотором пространстве (скажем, $\mathbb{R}^2$) и что требуется найти точку `y`, для которой сумма расстояний до всех заданных точек минимальна. Иными словами, если `D` – функция расстояния, то нужно найти точку `y`, минимизирующую выражение $D(y, x_1) + D(y, x_2) + D(y, x_3) + \ldots$.

Оптимизатору необходима функция, которая принимает все заданные точки и точку-кандидат и для каждого x_i вычисляет $D(y, x_i)$. Звучит очень похоже на операцию map-reduce, и функция `apop_map_sum` пользуется этим сходством (она даже распараллеливает обработку с помощью OpenMP). Структура `apop_data` предоставляет единое средство задать набор точек `x`, суммарное расстояние до которых оптимизируется. Кстати, физики и GSL предпочитают минимизацию, а экономисты и Apophenia – максимизацию. Это различие легко преодолевается добавлением

знака минус: вместо минимизации суммы расстояний следует максимизировать противоположное значение.

Процедура оптимизации довольно сложна (какова размерность пространства поиска? где оптимизатору взять эталонный набор данных? какой алгоритм поиска применять?), поэтому функция apop_estimate принимает структуру apop_model, содержащую поля для функции и необходимой дополнительной информации. Может показаться странным называть минимизатор суммы расстояний моделью, но многое из того, что мы считаем статистическими моделями (линейная регрессия, метод опорных векторов, имитационное моделирование и т. д.), обсчитывается именно так: принимают данные, находят оптимум при заданной целевой функции и говорят, что этот оптимум является наиболее вероятным набором параметров для этой модели при указанных данных.

В примере 13.4 показана полная процедура: написание функции расстояния, обертывание ее и всех необходимых данных структурой apop_model и занимающее всего одну строку обращение к функции apop_estimate, которая выполняет оптимизацию и возвращает структуру модели, в которой описывается точка, минимизирующая сумму расстояний до заданных точек.

Пример 13.4 ❖ Поиск точки, в которой достигает минимума сумма расстояний до заданных точек (gsl_distance.c)

```
#include <apop.h>

double one_dist(gsl_vector *v1, void *v2){
  return apop_vector_distance(v1, v2);
}

long double distance(apop_data *data, apop_model *model){
  gsl_vector *target = model->parameters->vector;
  return -apop_map_sum(data, .fn_vp=one_dist, .param=target, .part='r');    ❶
}

apop_model *min_distance= &(apop_model){
  .name="Minimum distance to a set of input points.", .p=distance,          ❷
  .vsize=-1};

int main(){
  apop_data *locations = apop_data_falloc((5, 2),                           ❸
            1.1, 2.2,
            4.8, 7.4,
            2.9, 8.6,
            -1.3, 3.7,
            2.9, 1.1);
  Apop_model_add_group(min_distance, apop_mle, .method="NM simplex",        ❹
                                         .tolerance=1e-5);
  Apop_model_add_group(min_distance, apop_parts_wanted);                    ❺
  apop_model *est=apop_estimate(locations, min_distance);                   ❻
  apop_model_show(est);
}
```

❶ Полагая `.part='r'`, я говорю, что хочу применить функцию `one_dist` к каждой строке входного набора данных. Знак минус присутствует, потому что мы применяем систему, выполняющую максимизацию, для минимизации расстояния.
❷ Элемент `.vsize` напоминает, что под капотом `apop_estimate` много чего делает. При этом выделяется память для элемента, содержащего параметры модели, а значение −1 означает, что количество параметров должно быть равно количеству столбцов в наборе данных.
❸ Первый аргумент функции `apop_data_falloc` – это список, содержащий количество точек и их размерность; затем идут сами данные: пять точек в двухмерном пространстве. См. раздел «Несколько списков» на стр. 216.
❹ В модель можно добавить группы настроек, полезные для различных функций. В этой строке мы добавляем сведения, касающиеся оптимизации: использовать симплекс-метод Нелдера-Мида и стремиться, чтобы погрешность составляла менее 1e−5. При наличии параметра `.verbose=1` будет выводиться дополнительная информация о каждой итерации поиска оптимума.
❺ Поскольку библиотека Apophenia предназначена для статистических расчетов, она вычисляет ковариацию и другие статистические характеристики параметров. Если эти вещи вас не интересуют, то их вычисление было бы пустой тратой времени, поэтому мы задаем пустую группу `apop_parts_wanted`, показывая, что никакая дополнительная информация не нужна.
❻ Теперь все подготовлено и можно выполнить собственно оптимизацию: найти точку, в которой функция `min_distance` принимает минимальное значение при заданных точках.

SQLite

Структурированный язык запросов (SQL) позволяет человеку взаимодействовать с базой данных. Поскольку база данных обычно хранится на диске, она может быть сколь угодно большой. Язык SQL среди прочего обладает двумя свойствами, особенно важными при работе с большими наборами данных: умение выделять подмножество данных и соединение наборов данных.

Я не стану углубляться в описание SQL, поскольку этой теме посвящены толстенные тома. Если мне будет позволено процитировать самого себя, то в книге [Klemens 2008] имеется глава, посвященная SQL и работе с ним из C. Можете также задать поисковику запрос *sql tutorial*. Основы довольно просты. Я сделаю акцент на знакомстве с библиотекой SQLite.

SQLite реализована в виде одного C-файла и одного заголовка. В этот файл входят анализатор SQL-запросов, различные внутренние структуры и функции для доступа к данным в файле на диске и несколько десятков интерфейсных функций для взаимодействия с базой данных. Скачайте файл, распакуйте его в каталог проекта, добавьте `sqlite3.o` в цель `objects` своего файла makefile – и вы получите полнофункциональный движок базы данных.

Вам придется иметь дело всего с несколькими функциями: открыть базу данных, закрыть базу данных, послать запрос и получить строки данных из базы.

Вот как выглядят функции открытия и закрытия базы данных:

```
sqlite3 *db=NULL; // глобальный описатель базы данных
int db_open(char *filename){
  if (filename)  sqlite3_open(filename, &db);
  else           sqlite3_open(":memory:", &db);
  if (!db) {printf("Не удалось открыть базу данных.\n"); return 1;}
  return 0;
}

// функция закрытия базы данных совсем проста:
sqlite3_close(db);
```

Я предпочитаю объявлять один глобальный описатель базы данных. Если нужно открыть несколько баз данных, то можно воспользоваться командой SQL attach. SQL-команды для доступа к таблице в такой присоединенной базе данных выглядят примерно так:

```
attach "diskdata.db" as diskdb;
create index diskdb.index1 on diskdb.tab1(col1);
select * from diskdb.tab1 where col1=27;
```

Если первый описатель относится к базе данных в памяти, а присоединенные базы данных находятся на диске, то нужно явно указывать, какие новые таблицы и индексы записываются на диск; любой объект, относительно которого нет таких указаний, будет временной таблицей в памяти – доступ к ней быстрее, но при завершении программы она пропадает. Если вы по оплошности создали таблицу в памяти, то впоследствии ее можно будет записать на диск командой вида create table diskdb.saved_table as select * from table_in_memory.

Запросы

Ниже приведен макрос для отправки базе данных SQL-запроса, не возвращающего данных. Например, команды attach и create index просят базу выполнить определенные действия, но ничего не возвращают.

```
#define ERRCHECK {if (err!=NULL) {printf("%s\n",err); return 0;}}

#define query(...){char *query; asprintf(&query, __VA_ARGS__);  \
                   char *err=NULL;                               \
                   sqlite3_exec(db, query, NULL,NULL, &err);     \
                   ERRCHECK                                      \
                   free(query); free(err);}
```

Макрос ERRCHECK взят непосредственно из руководства по SQLite. Я обернул обращение к sqlite3_exec макросом, чтобы можно было использовать такие конструкции:

```
for (int i=0; i< col_ct; i++)
  query("create index idx%i on data(col%i)", i, i);
```

Построение строки запросов в стиле printf – обычное дело при работе с SQL из С. Как правило, большинство запросов в программе строится как раз динамически. Но у этого подхода есть недостаток: в SQL-операторе like и в форматной

строке printf знак% имеет специальный смысл, поэтому вызов query("select * from data where col1 like 'p%%nts'") завершится с ошибкой, так как printf полагает, что конструкция%% предназначена для нее. Правильный вариант выглядит так: query("%s", "select * from data where col1 like 'p%%nts'"). Тем не менее динамическое построение запросов настолько употребительно, что можно смириться с мелким неудобством в виде добавления лишней строки%s для статических запросов.

Чтобы получить от SQLite данные, необходимо задать функцию обратного вызова (см. раздел «Функции с обобщенными входными параметрами»). Вот пример функции, которая выводит результаты на экран.

```
int the_callback(void *ignore_this, int argc, char **argv, char **column){
  for (int i=0; i< argc; i++)
     printf("%s,\t", argv[i]);
  printf("\n");
  return 0;
}

#define query_to_screen(...){                          \
  char *query; asprintf(&query, __VA_ARGS__);          \
  char *err=NULL;                                      \
  sqlite3_exec(db, query, the_callback, NULL, &err); \
  ERRCHECK                                             \
  free(query); free(err);}
```

Входные параметры функции обратного вызова напоминают параметры main: имеется параметр argv – список строк длиной argc. Имена столбцов (также список строк длиной argc) передаются в параметре column. Для вывода на экран как-то преобразовывать строки не нужно, так что особых трудностей не возникает. А вот как выглядит функция, заполняющая массив:

```
typedef {
  double *data;
  int rows, cols;
} array_w_size;

int the_callback(void *array_in, int argc, char **argv, char **column){
  array_w_size *array = array_in;
  *array = realloc(&array->data, sizeof(double)*(++(array->rows))*argc);
  array->cols=argc;
  for (int i=0; i< argc; i++)
     array->data[(array->rows-1)*argc + i] = atof(argv[i]);
}

#define query_to_array(a, ...){                       \
  char *query; asprintf(&query, __VA_ARGS__);         \
  char *err=NULL;                                     \
  sqlite3_exec(db, query, the_callback, a, &err);     \
  ERRCHECK                                            \
  free(query); free(err);}

// пример использования:
array_w_size untrustworthy;
query_to_array(&untrustworthy, "select * from people where age >%i", 30);
```

Проблема возникает тогда, когда имеются как строковые, так и числовые данные. Код, в котором обрабатывается случай таких смешанных данных, занимает в вышеупомянутой библиотеке Apophenia одну или две страницы.

Тем не менее достойно восхищения, что этих двух коротких фрагментов кода вкупе с двумя файлами SQLite и небольшой модификацией строки `objects` в makefile достаточно для включения в программу полнофункциональной базы данных SQL.

libxml и cURL

cURL – это написанная на C библиотека, поддерживающая длинный список протоколов Интернета, в том числе HTTP, HTTPS, POP3, Telnet, SCP и, конечно, Gopher. Если вам понадобится организовать взаимодействие с сервером, то, скорее всего, libcURL поможет. Как будет видно из следующего примера, библиотека предоставляет простой интерфейс, требующий от программиста задать всего несколько переменных и затем установить соединение.

Раз уж мы заговорили об Интернете, где повсеместно употребляются языки XML и HTML, уместно будет познакомиться заодно и с библиотекой libxml2.

Расширяемый язык разметки (XML) применяется для форматирования простых текстовых файлов, но по существу он определяет древовидную структуру. В верхней части рис. 13.1 показан фрагмент данных в формате XML, в котором практически невозможно разобраться; в нижней половине те же данные представ-

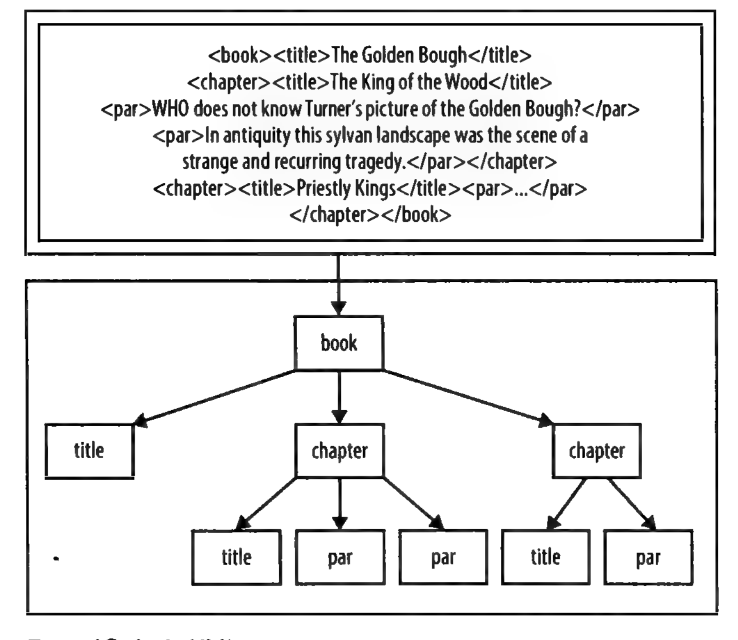


Рис. 13.1 ❖ XML-документ и представляющее его дерево

лены в виде дерева. Работать с подробно размеченным деревом сравнительно просто: можно начать с корневого узла (возвращаемого функцией `xmlDocGetRootElement`) и рекурсивно обойти все элементы. Или получить все элементы с тегом `par`. Или все элементы с тегом `title`, являющиеся потомками второй главы. И так далее. В следующем примере путь `//item/title` отбирает все элементы `title` с родителем `item`, где бы они ни находились в дереве.

Таким образом, библиотека libxml2 говорит на языке размеченных деревьев, а объектами в ней являются представления документа, узлы и списки узлов.

В примере 13.5 приведен полный пример. Я документировал его с помощью Doxygen (см. раздел «Встроенная документация»), потому он и оказался таким длинным, зато код сам объясняет себя. Если вы обычно пропускаете длинные куски кода, попробуйте все-таки задержаться и оценить, насколько эта программа понятна. Если у вас под рукой есть Doxygen, можете сгенерировать документацию и ознакомиться с ней в браузере.

Пример 13.5 ❖ Разбор ленты новостей газеты «Нью-Йорк таймс» и преобразование ее в более простой формат (nyt_feed.c)

```
/** \file

Программа чтения ленты главных новостей Нью-Йорк таймс и создания
более простой HTML-страницы. */
#include <stdio.h>
#include <curl/curl.h>
#include <libxml2/libxml/xpath.h>
#include "stopif.h"

/** \mainpage
Начальная страница сайта газеты Нью-Йорк таймс, выполненная в кричащем
стиле, содержит несколько главных новостей и разделов в попытке завладеть
вниманием читателя. Присутствуют различные схемы форматирования и даже
фотографии -- <em>цветные</em>.

Эта программа читает новостную RSS-ленту газеты и формирует простой список
в формате HTML. Затем вы можете щелкнуть по заинтересовавшей вас ссылке.

Замечания о компиляции см. на странице \ref компиляция.
*/

/** \page компиляция Компиляция программы

Сохраните следующий код в \c makefile.

Обратите внимание, что в состав cURL входит программа \c curl-config,
которая ведет себя как \c pkg-config, но с учетом специфики cURL.

\code
CFLAGS =-g -Wall -O3 `curl-config --cflags` -I/usr/include/libxml2
LDLIBS=`curl-config --libs ` -lxml2 -lpthread
CC=c99

nyt_feed:
```

```
\endcode

Сохранив makefile, выполните команду <tt>make nyt_feed</tt> для компиляции.

Разумеется, должны быть установлены пакеты разработчика для libcurl и
libxml2.
*/

// В код встроена документация в формате Doxygen. Символ < указывает на
// предыдущий документируемый текст.
char *rss_url = "http://rss.nytimes.com/services/xml/rss/nyt/HomePage.xml";
                                  /**< URL RSS-ленты NYT. */
char *rssfile = "nytimes_feeds.rss";
                 /**< Локальный файл, в который пишется RSS-лента.*/
char *outfile = "now.html";
               /**< выходной файл, открываемый в браузере.*/

/** Вывести список главных новостей в формате HTML в outfile, затирая
предыдущий вариант.
\param urls Список url-адресов. Должен быть отличен от NULL.
\param titles Список заголовков, также должен быть отличен от NULL. Если
длина списка \c urls или списка \c titles равна \c NULL, программа
грохнется.
*/
void print_to_html(xmlXPathObjectPtr urls, xmlXPathObjectPtr titles){
  FILE *f = fopen(outfile, "w");
  for (int i=0; i< titles->nodesetval->nodeNr; i++)
     fprintf(f, "<a href=\"%s\">%s</a><br>\n"
              , xmlNodeGetContent(urls->nodesetval->nodeTab[i])
              , xmlNodeGetContent(titles->nodesetval->nodeTab[i]));
  fclose(f);
}

/** Разобрать RSS-ленту на диске. Функция разбирает XML, затем находит все
узлы, удовлетворяющие выражению XPath для элементов заголовков, и все узлы,
удовлетворяющие выражению XPath для элементов ссылок. Эти узлы далее
записываются в outfile.

\param infile Файл, содержащий RSS-ленту.
*/
int parse(char const *infile){
  const xmlChar *titlepath= (xmlChar*)"//item/title";
  const xmlChar *linkpath= (xmlChar*)"//item/link";

  xmlDocPtr doc = xmlParseFile(infile);
  Stopif(!doc, return -1, "Ошибка при разборе файла \"%s\"\n", infile);

  xmlXPathContextPtr context = xmlXPathNewContext(doc);
  Stopif(!context, return -2, "Ошибка при создании контекста XPath\n");

  xmlXPathObjectPtr titles = xmlXPathEvalExpression(titlepath, context);
  xmlXPathObjectPtr urls = xmlXPathEvalExpression(linkpath, context);
  Stopif(!titles || !urls, return -3, "ошибка в Xpath '//item/title' "
                                  "или в '//item/link'.");
```

```
  print_to_html(urls, titles);

  xmlXPathFreeObject(titles);
  xmlXPathFreeObject(urls);
  xmlXPathFreeContext(context);
  xmlFreeDoc(doc);

  return 0;
}

/** Используем простой интерфейс cURL для скачивания RSS-ленты.
\param url URL RSS-ленты Нью-Йорк таймс. Все адреса, перечисленные в
           \url http://www.nytimes.com/services/xml/rss/nyt/, должны
           работать.
\param outfile Файл главных новостей, в который производится запись. Сначала
в этом файле сохраняется RSS-лента, а затем вместо нее записывается короткий
список ссылок.
\return 1==OK, 0==ошибка.
*/
int get_rss(char const *url, char const *outfile){
  FILE *feedfile = fopen(outfile, "w");
  if (!feedfile) return -1;

  CURL *curl = curl_easy_init();
  if(!curl) return -1;
  curl_easy_setopt(curl, CURLOPT_URL, url);
  curl_easy_setopt(curl, CURLOPT_WRITEDATA, feedfile);
  CURLcode res = curl_easy_perform(curl);
  if (res) return -1;

  curl_easy_cleanup(curl);
  fclose(feedfile);
  return 0;
}

int main(void) {
  Stopif(get_rss(rss_url, rssfile), return 1,
         "не удалось скачать%s в%s.\n",
         rss_url, rssfile);
  parse(rssfile);
  printf("Главные новсти записаны в%s. Откройте этот файл в браузере.\n",
         outfile);
}
```

Эпилог

Зажги новую спичку, начни сначала.

— Боб Дилан
на закрытии фестиваля
фолк-музыки 1965 года в Ньюпорте,
песня «It's All Over Now Baby Blue»

«Но постой-ка! – слышу я возгласы. – Ты же обещал, что я смогу использовать библиотеки, чтобы облегчить себе работу. Но я – специалист в своей области – искал повсюду, но так и не нашел библиотеки, которая мне нужна!»

Если это вы сетовали, то самое время раскрыть секретную цель, которую я преследовал при написании этой книги: будучи пользователем C, я хочу, чтобы как можно больше людей писали библиотеки, которыми я смог бы воспользоваться. Если вы дочитали до этого места, то знаете, как писать современный код, в котором используются библиотеки, как создать набор функций, предоставляющих интерфейс к простому объекту, как сделать удобный интерфейс, как документировать свой код, чтобы другие могли его использовать, какие имеются средства тестирования, как работать с репозиторием Git, чтобы ваши соавторы могли внести свой вклад, и как собрать код в пакет для пользователей, имеющих Autotools. C – основа современных вычислений, поэтому если вы оформите решение задачи в виде кода на C, но оно будет доступно на самых разных платформах.

Панк-рок – это вид искусства «сделай сам». Общепризнано, что музыка пишется такими же людьми, как мы, и что не требуется специального разрешения от какого-то наблюдательного комитета для создания и распространения своих творений во всем мире. И у нас есть инструменты, позволяющие это делать.

Приложение

Основные сведения о языке C

В этом приложении рассматриваются основы языка. Оно предназначено не для всех.

- Если у вас уже есть опыт программирования на каком-нибудь распространенном скриптовом языке, например Python, Ruby или Visual Basic, то это приложение как раз для вас. Я не собираюсь объяснять, что такое переменные, функции, циклы и прочие базовые конструкции, поэтому сами названия разделов сообщают о существенных различиях между C и типичными скриптовыми языками.
- Если вы изучали C давно и успели изрядно подзабыть, то беглый просмотр этого приложения поможет вспомнить тонкости, отличающие C от других языков.
- Если вы работаете с C постоянно, то не тратьте времени на это приложение. Быть может, даже первые главы части II стоит пропустить или проглядеть лишь мельком, поскольку они посвящены типичным ошибкам и недопониманию базовых основ языка.

Не думайте, что, прочитав это пособие, вы станете экспертом по C, – не существует никакой замены практическому опыту работы. Но вы все же сможете приступить к чтению части II и разобраться в нюансах и полезных обычаях языка.

Структура

Я начну руководство так же, как это сделали Керниган и Ричи в своей знаменитой книге 1978 года: с программы, которая здоровается с миром.

```
//tutorial/hello.c
#include <stdio.h>

int main(){
    printf("Hello, world.\n");
}
```

Два знака косой черты в первой строке обозначают комментарий, игнорируемый компилятором. Все примеры кода в этом приложении, в начале которых указано имя файла, можно скачать с сайта **https://github.com/b-k/21st-Century-Examples**.

Даже этот крохотный фрагмент многое говорит о языке C. С точки зрения структуры, в программе на C можно выделить следующие элементы:

- директива препроцессора, например `#include <stdio.h>`;
- объявление переменной или типа (в этой программе нет ни того, ни другого);
- блок функции, например `main`, содержащий выражения, подлежащие вычислению (например, `printf`).

Но прежде чем переходить к деталям определения и использования директив препроцессора, объявлений, блоков и выражений, нужно понять, как эту программу запустить, чтобы компьютер мог нас поприветствовать.

В С необходим этап компиляции, состоящий из одной команды

Скриптовый язык подразумевает наличие программы, которая интерпретирует текст скрипта; для С имеется компилятор, который принимает исходный текст программы и порождает файл, исполняемый непосредственно операционной системой.

Если на вашей машине нет компилятора, прочитайте раздел «Работа с менеджером пакетов», где описано, как его добыть. Короткий совет: попросите менеджер пакетов установить gcc или clang и make.

Итак, допустим, что компилятор и make установлены. Если вы сохранили приведенную выше программу в файле `hello.c`, то, для того чтобы заставить make запустить компилятор, достаточно такой команды:

```
make hello
```

В результате будет создан файл `hello`, который можно выполнить из командной строке или щелкнув по нему в файловом менеджере. Сделайте это и убедитесь, что печатается именно то, что ожидается.

В репозитории исходного кода имеется файл makefile, который содержит инструкции для `make`: какие задавать флаги компилятора. Принципы работы make и содержание makefile подробно обсуждаются в разделе «Работа с файлами makefile». А пока я упомяну только один флаг: `-Wall`. Он просит компилятор выводить все предупреждения о тех местах в коде, которые технически правильны, но, возможно, будут работать не так, как вы хотели. Эта техника называется *статическим анализом*, современные компиляторы владеют ей прекрасно. Таким образом, можно считать, что шаг компиляции – не просто бесполезная формальность, но еще и шанс перед запуском предложить свой код на рассмотрение команде ведущих мировых экспертов по С.

Если вы работаете с компилятором на платформе Мас, который не понимает флага `-Wall`, прочитайте в разделе «Несколько моих любимых флагов» замечание о том, как создать псевдоним для `gcc`.

Многие блогеры считают шаг компиляции обузой. Но если вы пользуетесь интегрированной средой разработки (IDE), то почти наверняка в ней есть кнопка «Откомпилировать и выполнить». Если вы считаете, что набрать `make yourprogram` в командной строке перед запуском `./yourprogram` – непосильный труд, то можете

написать псевдоним или скрипт оболочки. В совместимой с POSIX оболочке можно было бы определить функцию:

```
function crun { make $1 && ./$1; }
```

и использовать ее следующим образом:

```
crun hello
```

При этом будет произведена попытка компиляции, и если она пройдет успешно, то программа будет запущена.

Существует стандартная библиотека, это часть операционной системы

Современные программы редко бывают совсем автономными, чаще они компонуются с библиотеками общих функций, которые, возможно, используются во многих программах. Путь к библиотекам – это список каталогов на диске, в которых компилятор ищет библиотеки, подробнее см. раздел «Пути» в главе 1. Самой главной среди всех библиотек является стандартная библиотека, определенная в стандарте ISO C. Она настолько близка к универсальной доступности, насколько это вообще возможно для компьютерного кода. Именно в ней находится функция `printf`.

Существует препроцессор

Библиотеки хранятся в двоичном формате, они исполняются компьютером, но для человека представляют бессмысленный набор байтов. Если вы не обладаете сверхъестественной способностью читать двоичные файлы, то, глядя на откомпилированную библиотеку, не сможете понять, правильно ли вы используете функцию `printf`. Поэтому в дополнение к библиотеке имеются *файлы-заголовки*, содержащие текстовые объявления реализованных в библиотеке средств, а именно: для каждой функции приводятся описания ожидаемых входных параметров и возвращаемых значений. Если вы включите в свою программу подходящий заголовок, то компилятор сможет проверить совместимость использования всех функций, переменных и типов с тем, что ожидает библиотека.

Основная задача препроцессора заключается в том, чтобы заменять текст, заданный в директивах препроцессора (все они начинаются знаком #), другим текстом. Существует много иных применений (см. раздел «Приемы работы с препроцессором» в главе 8), но в этом приложении я ограничусь только включением файлов. Видя директиву

```
#include <stdio.h>
```

препроцессор подставляет вместо нее полный текст файла `stdio.h`. Угловые скобки в `<stdio.h>`, говорят, что файл нужно искать на пути включения (это не то же самое, что путь к библиотекам, см. раздел «Пути» в главе 1). Если файл находится в рабочем каталоге проекта, то нужно использовать форму `#include "myfile.h"`.

Суффикс .h в имени файла означает, что это заголовок. Заголовки содержат обычный текст, и компилятор не отличает их от других файлов с кодом, но принято помещать объявления именно в заголовки.

После завершения работы препроцессора почти все в коде оказывается либо объявлением переменной или типа, либо определением функции.

Существуют комментарии двух видов

```
/* Многострочный комментарий заключен между комбинациями символов
косая черта-звездочка и звездочка-косая черта. */
// Однострочный комментарий начинается после двух символов косой черты
// и продолжается до конца строки.
```

Нет ключевого слова print

Функция printf из стандартной библиотеки выводит текст на экран. У нее есть собственный мини-язык для точного указания, как должны печататься переменные. Не буду вдаваться в детальные объяснения, потому что исчерпывающие описания языка форматирования, применяемого в printf, можно найти в других местах (попробуйте набрать команду man 3 printf), да к тому же примеров достаточно как в этом руководстве, так и в основном тексте книги. Предложение на этом мини-языке состоит из обычного текста, в котором встречаются *маркеры подстановки переменных* и специальные коды для символов, не имеющих графического начертания, например знаков табуляции и перехода на новую строку. Ниже перечислены шесть элементов, которые попадаются в примерах использования функций из семейства printf:

- \n переход на новую строку;
- \t табулятор;
- %i вставить целое значение;
- %g вставить вещественное значение в общем формате;
- %s вставить строку текста;
- %% вставить один знак процента.

Объявления переменных

Объявления в C и в большинстве скриптовых языков существенно отличаются тем, что в скриптовых языках тип переменной и даже сам факт ее существования устанавливаются при первом использовании. Выше я уже говорил о том, что этап компиляции дает возможность еще до выполнения проверить, что код хотя бы теоретически будет делать то, что обещано; объявление типа любой переменной позволяет компилятору удостовериться, что код непротиворечив. Существует также синтаксис объявления функций и новых типов.

Любая переменная должна быть объявлена

В программе hello переменных не было, но ниже приведена программа, в которой объявляется несколько переменных и демонстрируется использование printf. Об-

ратите внимание на первый аргумент printf (*форматную строку*); в ней имеются три *подстановочных маркера* (*спецификаторы формата*), поэтому далее следуют три переменные, подставляемые вместо маркеров.

```
//tutorial/ten_pi.c
#include <stdio.h>

int main(){
    double pi= 3.14159265;  // кстати, в POSIX определена константа M_PI
                            // в файле math.h
    int count= 10;
    printf("%g *%i =%g.\n", pi, count, pi*count);
}
```

Эта программа выводит:

```
3.14159 * 10 = 31.4159.
```

В этой книге встречаются три базовых типа: int, double и char, соответственно, целое число, число с плавающей точкой двойной точности и символ.

Некоторые блогеры считают, что необходимость объявлять переменные – участь хуже смерти, но, как видно из примера, весь труд сводится к добавлению имени типа до первого использования переменной. А при чтении незнакомого кода наличие типа каждой переменной и признака первого использования – неплохие ориентиры.

Если есть несколько переменных одного типа, то их можно объявить в одной строке:

```
int count=10, count2, count3=30; // count2 не инициализирована.
```

Даже функции необходимо объявлять или определять

В определении функции описывается все, что она делает, как, например, в следующем тривиальном примере:

```
int add_two_ints(int a, int b){
    return a+b;
}
```

Эта функция принимает два целых числа, которые внутри функции называются a и b, и возвращает одно целое число – сумму a и b.

Можно вынести объявление в отдельное предложение, в котором указываются имя функции, типы входных параметров (в скобках) и тип возвращаемого значения:

```
int add_two_ints(int a, int b);
```

Здесь не сказано, что в действительности делает функция add_two_ints, но и такого объявления компилятору достаточно, чтобы проверить правильность всех случаев употребления функции: что ей передается два целых числа и что результатом является тоже целое. Объявление функции, как и любое другое, может находиться прямо в файле с кодом или в файле-заголовке, который включается директивой вида #include "mydeclarations.h".

Блоком называется участок кода, заключенный в фигурные скобки. Следовательно, определение функции представляет собой объявление, за которым следует один блок, исполняемый при вызове функции.

Если полное определение функции предшествует ее первому использованию, то у компилятора есть все необходимое для проверки корректности, и отдельное объявление излишне. Поэтому программы на С чаще всего пишутся и читаются «снизу вверх», то есть функция main оказывается последней в файле, после определения всех функций, вызываемых из main, а перед этими функциями находятся определения тех, которые вызываются из них, и т. д. до заголовков в начале файла, где объявлены все используемые библиотечные функции.

Кстати говоря, функция может иметь тип void, означающий, что она не возвращает ничего. Это полезно в случае функций, которые ничего не возвращают и не изменяют входных параметров, но имеют побочные эффекты. Например, ниже приведена программа, состоящая в основном из функции, которая выводит в файл (он будет создан на диске) сообщения об ошибках в фиксированном формате. Тип FILE и все сопутствующие ему функции объявлены в заголовке stdio.h. Почему строка текста объявлена в виде переменной типа char*, я объясню ниже.

```
//tutorial/error_print.c
#include <stdio.h>

void error_print(FILE *ef, int error_code, char *msg){
  fprintf(ef, "Произошла ошибка #%i:%s.\n", error_code, msg);
}

int main(){
  FILE *error_file = fopen("example_error_file", "w"); //открыть для записи
  error_print(error_file, 37, "Вся карма вышла");
}
```

Базовые типы можно агрегировать в массивы и структуры

Как работать, имея только три базовых типа? Путем создания составных типов: массивов, содержащих элементы одного и того же типа, и структур, содержащих элементы разных типов.

В следующей программе объявляются список из десяти целых чисел и строка из 20 символов, после чего используются части того и другого.

```
//tutorial/item_seven.c
#include <stdio.h>

int intlist[10];

int main(){
    int len=20;
    char string[len];

    intlist[7] = 7;
    snprintf(string, 20, "Элемент 7 равен%i.", intlist[7]);
    printf("строка содержит: <<%s>>\n", string);
}
```

Функция snprintf выводит в строку не более указанного количества символов, синтаксис форматной строки такой же, как у функции printf. Подробнее о работе со строками и о том, почему переменную intlist можно объявить вне функции, а переменная string должна быть объявлена внутри, я расскажу ниже.

Индекс – это *смещение* от начала массива. Первый элемент находится в самом начале массива, поэтому обозначается intlist[0]; последним элементом массива из десяти элементов будет intlist[9]. Это еще одна причина паники и ожесточенных перепалок в Сети, но у этого соглашения есть смысл.

У некоторых композиторов есть нулевые симфонии (Брукнер, Шнитке). Но, как правило, мы пользуемся порядковыми числительными – *первый, второй, седьмой*, и это вступает в противоречие со схемой нумерации по смещению: седьмой элемент массива обозначается intlist[6]. Поэтому я стараюсь пользоваться другой языковой конструкцией: *элемент 6 массива*.

По причинам, которые станут ясны позже, тип массива можно обозначать также звездочкой:

```
int *intlist;
```

Пример вы уже видели выше, при объявлении последовательности символов в виде char *msg.

Можно определять новые структурные типы

Данные разных типов можно объединить в структуру (обозначается ключевым словом struct) и рассматривать как единое целое. В примере ниже объявляется и используется тип ratio_s, описывающий дробь, которая имеет числитель, знаменатель и десятичное значение. Псевдоним типа – typedef – это по существу список объявлений в фигурных скобках.

При работе со структурами бросается в глаза изобилие точек: если дана структура r типа ratio_s, то хранящийся в ней числитель обозначается r.numerator. Выражение (double)den – это приведение типа, в результате которого целая переменная den преобразуется в тип double (зачем это нужно, объясняется ниже). Синтаксис инициализации новой структуры выглядит как приведение типа: сначала идет заключенное в круглые скобки имя структурного типа, а затем начинающиеся с точки элементы этого типа, заключенные в фигурные скобки. Существуют и более краткие (и оттого менее понятные) способы инициализации структуры.

```
//tutorial/ratio_s.c
#include <stdio.h>

typedef struct {
  int numerator, denominator;
  double value;
} ratio_s;

ratio_s new_ratio(int num, int den){
  return (ratio_s){.numerator=num, .denominator=den,
                   .value=num/(double)den};
```

```
}

void print_ratio(ratio_s r){
  printf("%i/%i =%g\n", r.numerator, r.denominator, r.value);
}

ratio_s ratio_add(ratio_s left, ratio_s right){
  return (ratio_s){
     .numerator=left.numerator*right.denominator
            + right.numerator*left.denominator,
     .denominator=left.denominator * right.denominator,
     .value=left.value + right.value
  };
}

int main(){
  ratio_s twothirds= new_ratio(2, 3);
  ratio_s aquarter= new_ratio(1, 4);
  print_ratio(twothirds);
  print_ratio(aquarter);
  print_ratio(ratio_add(twothirds, aquarter));
}
```

Можно узнать размер типа

Оператор sizeof принимает имя типа и сообщает, сколько памяти будет занимать экземпляр этого типа. Иногда это удобно.

Следующая короткая программа сравнивает размеры двух int и double с размером определенного выше типа ratio_s. Спецификатор формата%zu в форматной строке printf предназначен исключительно для печати результата sizeof.

```
//tutorial/sizeof.c
#include <stdio.h>

typedef struct {
  int numerator, denominator;
  double value;
} ratio_s;

int main(){
  printf("размер двух int:%zu\n", 2*sizeof(int));
  printf("размер двух int:%zu\n", sizeof(int[2]));
  printf("размер double:%zu\n", sizeof(double));
  printf("размер структуры ratio_s:%zu\n", sizeof(ratio_s));
}
```

Не существует специального типа строки

Оба целых числа, 5100 и 51, занимают sizeof(int) байтов в памяти. Но строки "Hi" и "Hello" содержат разное число символов. В скриптовых языках обычно существует специальный тип строки, рассчитанный на управление списками символов неопределенной длины. В С строка – это массив элементов типа char, ясно и просто.

Конец строки обозначается символом NUL, имеющим значение '\0', хотя он никогда не печатается; заботу о нем обычно берут на себя библиотечные функции. (Отметим, что символы заключаются в одиночные кавычки, например 'x', а строки – в двойные кавычки, например "xx"; строка из одного символа также заключается в двойные кавычки – "x"). Функция strlen(mystring) подсчитывает количество символов до нуля, не включая последнего. Но эта величина вовсе необязательно равна объему памяти, выделенной для строки; никто не мешает объявить char pants[1000] = "trousers", хотя при этом 991 байт после нуля оказывается потрачен впустую.

Сама природа строк делает возможными кое-какие забавные трюки. Если есть объявление

```
char* str="Hello";
```

то строку Hello можно превратить в Hell, вставив символ NUL:

```
str[4]='\0';
```

Но обычно работа со строками сводится к вызову той или иной библиотечной функции, которая манипулирует байтами. Вот несколько наиболее распространенных:

```
#include <string.h>
char *str1 = "hello", str2[100];
strlen(str1); // получить длину строки без учета '\0'
strncpy(str2, 100, str1); // скопировать не более 100 байтов из str1 в str2
strncat(str2, 100, str1); // добавить не более 100 байтов из str1 в конец str2
strcmp(str1, str2); // str1 и str2 одинаковы?
snprintf(str2, 100, "str1 содержит:%s", str1); // вывести в строку, как выше
```

В главе 9 обсуждается еще несколько функций, облегчающих работу со строками. При наличии качественных функций эта деятельность может снова стать приятной.

Функции и выражения

Правила видимости в С очень просты

Область видимости переменной – это та часть программы, в которой ее можно использовать.

Если переменная объявлена вне функции, то ее можно использовать в любом выражении от точки объявления до конца файла. Эта переменная доступна любой определенной в файле функции. Такие переменные инициализируются в начале работы программы и существуют вплоть до ее завершения. Они называются статическими, наверное, потому что занимают одно и то же место в памяти в течение всего времени работы программы.

Если переменная объявлена внутри блока (в том числе блока, содержащего определение функции), то она создается в точке объявления и уничтожается по достижении фигурной скобки, закрывающей этот блок.

Дополнительные замечания о статических переменных, в том числе о том, как получить переменную с постоянным временем жизни, видимую только внутри одной функции, см. в разделе «Переменные для хранения постоянного состояния» в главе 6.

Функция main имеет особый смысл

Первое, что делается после запуска программы, – инициализация глобальных переменных с областью видимости файла (см. выше). Пока еще никакие математические вычисления невозможны, поэтому таким переменным можно присвоить либо константное значение (если объявление имеет вид `int gv=24;`), либо значение нуль по умолчанию (в случае объявления вида `int gv;`).

В скриптовых языках обычно некоторые предложения находятся внутри функций, а некоторые – в основном теле скрипта, вне любой функции. В C любое вычисляемое выражение должно находиться в теле какой-то функции, а точкой входа в программу, где начинаются вычисления, является функция `main`. В примере функции `snprintf` выше массив длиной `len` должен был находиться внутри `main`, потому что получение значения `len` – слишком сложное действие для этапа инициализации программы.

Поскольку функция `main` вызывается операционной системой, она должна быть объявлена одним из двух допустимых способов:

```
int main(void);
// или, что то же самое
int main();
```

либо

```
int main(int, char**)
// и эти аргументы принято называть следующим образом:
int main(int argc, char** argv)
```

Примеры первого рода, когда ничего не подается на вход и возвращается одно целое число, мы уже видели. Это число интерпретируется как код ошибки: считается, что программа завершилась успешно, если код равен 0, и с ошибкой – в любом другом случае. Этот обычай укоренился настолько прочно, что в стандарте C даже говорится о том, что наличие предложения `return 0;` в конце `main` неявно подразумевается (см. раздел «Ни к чему явно возвращать значение из main» в главе 7). Простой случай объявления второго рода имеется в примере 8.6.

Большая часть работы программы на C сводится к вычислению выражений

Итак, глобальные переменные инициализированы, операционная система подготовила аргументы для `main`, и программа приступает к выполнению кода в блоке функции `main`.

Начиная с этого момента, все ее действия ограничиваются объявлением локальных переменных, управлением потоком выполнения (ветвление в предложении if-else, выполнение итераций цикла) и вычислением выражений.

Возвращаясь к приведенному ранее примеру, предположим, что система должна выполнить следующие предложения:

```
int a1, two_times;
a1 = (2+3)*7;
two_times = add_two_ints(a1, a1);
```

После объявлений идет строка `a1=(2+3)*7`, где требуется сначала вычислить выражение `(2+3)`, вместо которого можно подставить его значение 5, а затем вычислить выражение `5*7`, которое можно заменить значением 35. Именно так поступают люди, сталкиваясь с подобным выражением, но C идет по пути «вычисления и подстановки» гораздо дальше.

При вычислении выражения `a1=35` происходят две вещи. Первая – замена выражения его значением: 35. Вторая – побочный эффект, заключающийся в изменении состояния: значение переменной `a1` изменяется на 35. Существуют языки, стремящиеся к вычислениям без побочных эффектов, в них единственный результат вычисления – замена одного выражения другим. Но в C разрешены побочные эффекты вычислений, выражающиеся в изменении состояния. Еще с одним примером мы уже встречались неоднократно: при вычислении `printf("hello\n")` выражение заменяется нулем в случае успеха, но у него есть и побочный эффект: изменение изображения на экране.

После выполнения всех подстановок строка принимает вид `35;`. Когда больше вычислять нечего, система переходит к следующей строке.

При вычислении функций используются копии входных аргументов

В строке `two_times = add_two_ints(a1, a1)` требуется сначала дважды вычислить `a1`, затем вычислить результат функции `add_two_ints` с двумя ранее вычисленными аргументами, 35 и 35. Для этого функции передается *копия* значения `a1`, а не само `a1`. Это означает, что никаким способом функция не может изменить значение самой переменной `a1`. Если в коде функции встречаются предложения, модифицирующие входной аргумент, то надо понимать, что в действительности модифицируется не сам аргумент, а его копия. Что делать, если все-таки необходимо изменить переданные функции переменные, мы обсудим ниже.

Выражения заканчиваются точкой с запятой

Да, в C каждое выражение должно заканчиваться точкой с запятой. С точки зрения стиля, это спорное решение, зато оно позволяет размещать переходы на новую строку, дополнительные пробелы и знаки табуляции всюду, где это способствует повышения удобочитаемости программы.

Есть много сокращенных способов записи арифметических операций

В C есть ряд удобных сокращенных способов выразить операцию изменения переменной. Выражения `x=x+3` и `x=x/3` можно сократить соответственно до `x+=3` и `x/=3`. Увеличение переменной на единицу – настолько частая операция, что предложено

даже два ее варианта. У обоих выражений x++ и ++x имеется один и тот же побочный эффект: увеличение x на единицу (инкремент), но при вычислении x++ выражение заменяется значением x до инкремента, а при вычислении ++x – значением x после инкремента, то есть значением x+1.

```
x++;      // инкремент x. Значение равно x.
++x;      // инкремент x. Значение равно x+1.
x--;      // декремент x. Значение равно x.
--x;      // декремент x. Значение равно x-1.
x+=3;     // прибавить 3 к x.
x-=7;     // вычесть 7 из x.
x*=2;     // умножить x на два.
x/=2;     // разделить x на два.
x%=2;     // заменить x на x%2.
```

В С понятие истины трактуется расширительно

Иногда нужно знать, является ли выражение истинным или ложным, например когда принимается решение, по какой ветви предложения if-else идти. В С нет ключевых слов true и false, хотя они обычно определяются, как описано на врезке «Истина и ложь» на стр. 191. Вместо этого принято соглашение: если выражение равно нулю (или символу '\0', или указателю NULL), то оно считается ложным, во всех остальных случаях – истинным.

Верно и обратное: значением любого из приведенных ниже выражений является 0 или 1:

```
!x        // не x
x==y      // x равно y
x != y    // x не равно y
x < y     // x меньше y
x <= y    // x меньше или равно y
x || y    // x или y
x && y    // x и y
x > y || y >= z // x больше y либо y больше или равно z
```

Например, если x принимает любое значение, отличное от нуля, то значением !x будет нуль, а значением !!x – единица.

Операторы && и || – «ленивые», они вычисляют лишь часть выражения, достаточную для установления истинности или ложности всего выражения. Например, рассмотрим выражение (a < 0 || sqrt(a) < 10). Вычисление квадратного корня из –1 привело бы к ошибке (см., однако, обсуждение поддержки комплексных чисел в С в разделе «_Generic» на стр. 274). Но если a==-1, то мы точно знаем, что выражение (a < 0 || sqrt(a) < 10) равно true, какое бы значение ни принимала его вторая часть. Поэтому sqrt(a) < 10 вообще не вычисляется, и беды удается избежать.

Результатом деления двух целых всегда является целое

Многие программисты стремятся по возможности избегать чисел с плавающей точкой, потому что вычисления с целыми числами производятся быстрее и не дают ошибок округления. В С для этой цели есть три разных оператора: вещественное деление, целочисленное деление и взятие остатка. Первые два выглядят одинаково.

```
//tutorial/divisions.c
#include <stdio.h>

int main(){
    printf("3./5=%g\n", 3./5);
    printf("3/5=%i\n", 3/5);
    printf("3%%5=%i\n", 3%5);
}
```

Результат получается такой:

```
3./5=0.6
3/5=0
3%5=3
```

Выражение `3.` – это вещественное число с плавающей точкой. Если числитель или знаменатель – вещественное число, то производится вещественное деление, и результатом является вещественное число. Если числитель и знаменатель – целые числа, то производится вещественное деление, но результат округляется с недостатком до целого. Оператор% возвращает остаток от деления.

Различие между вещественным и целочисленным делением и есть причина, по которой в примере с `new_ratio` было выполнено приведение типа знаменателя в выражении `num/(double)den`. Дополнительные сведения см. в разделе «Меньше приведений» в главе 7.

В С имеется тернарный условный оператор

Если `a` и `b` – простые выражения, то результатом вычисления выражения

```
x ? a : b
```

будет `a`, если `x` истинно, и `b`, если `x` ложно.

Раньше мне казалось, что такая запись непонятна, и к тому же такой оператор есть не во всех скриптовых языках, но со временем его полезность стала для меня очевидной. Поскольку это просто выражение, мы можем употреблять его где угодно, например:

```
//tutorial/sqrt.c
#include <math.h> // Здесь объявлена функция sqrt
#include <stdio.h>

int main(){
  double x = 49;
  printf("Квадратный корень из x равен%g.\n",
      x > 0 ? sqrt(x) : 0);
}
```

Ветвления и циклы несильно отличаются от других языков

Пожалуй, единственная особенность С в части предложений `if-else` – отсутствие ключевого слова `then`. Вычисляемое условие заключается в скобки, и если оно истинно, то вычисляется следующее выражение или блок. Вот несколько примеров.

```
//tutorial/if_else.c
#include <stdio.h>

int main(){
    if (6 == 9)
        printf("Шесть равно девяти.\n");

    int x=3;
    if (x==1)
        printf("Я знаю, чему равно x: единице.\n");
    else if (x==2)
        printf("x точно равно двумя.\n");
    else
        printf("x не равно ни единице, ни двум.\n");
}
```

В цикле while вычисление блока повторяется, пока заданное условие не станет ложным. Например, следующая программа приветствует пользователя десять раз:

```
//tutorial/while.c
#include <stdio.h>

int main(){
  int i=0;
  while (i < 10){
     printf("Hello #%i\n", i);
     i++;
  }
}
```

Если управляющее условие в скобках после ключевого слова while оказывается ложным уже при первом вычислении, то тело цикла не выполняется ни разу. Однако тело цикла do-while гарантированно выполняется хотя бы один раз:

```
//tutorial/do_while.c
#include <stdio.h>

void loops(int max){
  int i=0;
  do {
     printf("Hello #%i\n", i);
     i++;
  } while (i < max); // Обратите внимание на точку с запятой.
}

int main(){
  loops(3); // печатаются три приветствия
  loops(0); // печатается одно приветствие
}
```

Цикл for – просто компактная форма цикла while

Управление циклом while состоит из трех частей:

- инициализация (int i=0);
- проверка условия (i < 10);
- шаг итерации (i++).

В цикле for все три части собраны в одном месте. Следующий цикл for эквивалентен показанному выше циклу while:

```
//tutorial/for_loop.c
#include <stdio.h>

int main(){
  for (int i=0; i < 10; i++){
     printf("Hello #%i\n", i);
  }
}
```

Поскольку блок занимает всего одну строку, то даже фигурные скобки можно опустить, тогда получится:

```
//tutorial/for_loop2.c
#include <stdio.h>

int main(){
  for (int i=0; i < 10; i++) printf("Hello #%i\n", i);
}
```

Часто возникают сомнения по поводу «ошибки на единицу», когда программист хотел выполнить десять итераций, а получил девять или одиннадцать. В приведенном выше варианте (начать с i=0, проверять условие i < 10) правильно отсчитывается десять итераций, и это стереотипная форма при обходе массива. Например:

```
int len=10;
double array[len];
for (int i=0; i< len; i++) array[i] = 1./(i+1);
```

Не существует специального синтаксиса для обхода любой последовательности или применения операции к любому элементу массива (хотя такой синтаксис можно реализовать с помощью макросов или функций), поэтому конструкции вида (int i=0; i< 10; i++) будут встречаться вам очень часто.

С другой стороны, точно известно, что делать в разных ситуациях. Если требуется шаг 2, то нужно написать for (int i=0; i< 10; i+=2). Если нужно продолжать итерации, пока в массиве не встретится нулевой элемент, то пишем for (int i=0; array[i]!=0; i++). Любую часть цикла можно опустить. Так, если не требуется инициализировать новую переменную цикла, то допустимо написать так: for (; array[i]!=0; i++).

Указатели

Указатели на переменные иногда называют псевдонимами, ссылками или метками (хотя в С имеются не относящееся к указателям и редко используемое понятие метки; см. раздел «Метки, goto, switch и break» в главе 7).

В указателе на double находится не само значение типа double, а адрес области в памяти, где это значение хранится. Таким образом, мы получаем два имени для одного и того же объекта. Если этот объект изменится, то изменение будет видно

через оба имени. И в этом принципиальное отличие от полной копии объекта, поскольку изменение оригинала никак не отражается на копии.

Можно напрямую запросить блок памяти

Функция malloc выделяет память для использования в программе. Например, вот как можно выделить место для хранения 3000 целых чисел:

```
malloc(3000*sizeof(int));
```

Это первое упоминание о явном выделении памяти в настоящем руководстве, потому что во встречавшихся раньше объявлениях вида list[100] память выделялась автоматически. При выходе из области видимости, в которой находится такое объявление, автоматически выделенная память автоматически же и освобождается. Напротив, память, выделенная динамически с помощью malloc, остается занятой, пока ее не освободят вручную (или до конца программы). Есть ситуации, когда такая долговечность весьма желательна. Кроме того, размер массива нельзя изменить после инициализации, тогда как динамически выделенную память можно перераспределить. Другие различия между динамическим и автоматическим выделением памяти обсуждаются в разделе «Автоматическая, статическая и динамическая память» в главе 6.

Итак, область памяти выделена, но как к ней обратиться? На помощь приходят указатели, поскольку мы можем связать псевдоним с областью, выделенной malloc:

```
int *intspace = malloc(3000*sizeof(int));
```

Звездочка в объявлении (int *) означает, что объявляется указатель на область памяти.

Память – конечный ресурс, поэтому беспечное использование может в конечном итоге привести к нехватке памяти – ошибке, с которой все мы когда-то сталкивались. Чтобы вернуть память системе, используется функция free, например: free(intspace). Или можно просто дождаться конца программы, когда операционная система сама освободит всю занятую программой память.

Массивы – это просто блоки памяти, любой блок памяти можно использовать как массив

В главе 6 подробно обсуждаются сходства и различия массивов и указателей, но, безусловно, они имеют много общего.

Массив занимает в памяти непрерывный участок, в котором элементы одного типа расположены друг за другом. Если запрашивается элемент 6 массива, объявленного как int list[100], то система возьмет элемент, отстоящий от начала этого участка на 6*sizeof(int) байтов.

Поэтому нотация с использованием квадратных скобок, list[6], – просто обозначение смещения от позиции, занятой именованной переменной. Именно это нам и нужно при работе с массивом. Но, имея указатель на непрерывный участок

памяти, мы могли бы проделать те же самые операции поиска элемента и продвижения к следующему элементу вручную.

В следующем примере показано, как заполнить динамически выделенный массив и вывести его в файл. То же самое было бы проще сделать, если бы массив выделялся автоматически, но нам важна демонстрация возможности.

```
//tutorial/manual_memory.c
#include <stdlib.h> //malloc and free
#include <stdio.h>

int main(){
  int *intspace = malloc(3000*sizeof(int));
  for (int i=0; i < 3000; i++)
     intspace[i] = i;

  FILE *cf = fopen("counter_file", "w");
  for (int i=0; i < 3000; i++)
     fprintf(cf, "%i\n", intspace[i]);

  free(intspace);
  fclose(cf);
}
```

Память, выделенную с помощью `malloc`, можно спокойно использовать в программе, но она не инициализирована и может содержать мусор. Чтобы выделить и обнулить память, пользуйтесь такой функцией:

```
int *intspace = calloc(3000, sizeof(int));
```

Обратите внимание, что она принимает два аргумента, а `malloc` – только один.

Указатель на скаляр – это по существу массив с одним элементом

Допустим, имеется указатель `i` на целое число. Это массив длины 1, и если мы напишем `i[0]`, то система найдет область, на которую указывает `i`, и отступит от нее на 0 элементов – в точности так же, как для более длинных массивов.

Но человеку несвойственно рассматривать одиночное значение как массив длиной 1, поэтому для часто встречающегося случая одноэлементного массива существует специальное соглашение: всюду, кроме строки объявления, `i[0]` и `*i` эквивалентны. Правда, это может стать источником путаницы, потому что в строке объявления звездочка означает нечто совершенно иное. Существуют доводы в пользу такого соглашения (см. раздел «Виноваты звезды» в главе 6), ну а пока просто запомните, что звездочка в объявлении означает новый указатель, а в любой другой строке – значение, на которое указатель указывает.

В следующем примере в первый элемент массива записывается значение 7. В последней строке проверяется, что в нем действительно находится 7, и, если я ошибся, программа аварийно завершается.

```
//tutorial/assert.c
#include <assert.h>
```

```
int main(){
  int list[100];
  int *list2 = list; // list2 объявляется как указатель на int,
                     // указывающий на тот же блок памяти, что и list

  *list2 = 7;        // list2 - указатель на int, поэтому *list2 - int.

  assert(list[0] == 7);
}
```

Существует специальная нотация для доступа к полям структур по указателю

Пусть имеется объявление

```
ratio_s *pr;
```

Тогда мы знаем, что pr – указатель на ratio_s, а не сама ratio_s. В памяти для pr отведено столько места, сколько занимает указатель, а не вся структура ratio_s.

Получить числитель, хранящийся в структуре, на которую направлен указатель, можно с помощью выражения (*pr).numerator, поскольку (*pr) – это структура ratio_s, а точка обозначает доступ к полю структуры. Существует также стрелочная нотация, которая избавляет от эстетического неудобства использования комбинации скобок и звездочки. Например:

```
ratio_s *pr = malloc(sizeof(ratio_s));
pr->numerator = 3;
```

Оба варианта: pr->numerator и (*pr).numerator – в точности эквивалентны, но первый обычно предпочтительнее в силу большей наглядности.

Указатели позволяют изменять аргументы функции

Напомним, что функции передаются копии входных переменных, а не сами переменные. При выходе из функции копии уничтожаются, а оригиналы остаются неизменными.

Теперь предположим, что функции передается указатель. Копия указателя указывает туда же, куда оригинал. Ниже приведена простая программа, которая пользуется этой стратегией для модификации переменной, на которую указывает аргумент функции.

```
//tutorial/pointer_in.c
#include <stdlib.h>
#include <stdio.h>

void double_in(int *in){
  *in *= 2;
}

int main(){
  int *x = malloc(sizeof(int));
  *x = 10;
```

```
  double_in(x);
  printf("теперь x указывает на%i.\n", *x);
}
```

Функция double_in не изменяет in, но удваивает значение, на которое in указывает, *in. Поэтому значение x, переданное функции double_in по указателю, удвоилось.

Это решение широко распространено, поэтому вы встретите много функций, принимающих указатель, а не просто значение. Но что сделать, чтобы передать такой функции указатель на переменную, содержащую простое значение? Для этого предназначен оператор &, который возвращает адрес переменной. Стало быть, если x – переменная, то &x – указатель на эту переменную. Это позволяет упростить показанный выше код.

```
//tutorial/address_in.c
#include <stdlib.h>
#include <stdio.h>

void double_in(int *in){
  *in *= 2;
}

int main(){
  int x = 10;
  double_in(&x);
  printf("теперь x указывает на%i.\n", *x);
}
```

Любой объект где-то находится, и, значит, на него можно указать

Невозможно передать функцию в качестве аргумента другой функции. Не бывает массивов функций. Но можно передать функции указатель на функцию и создать массив указателей на функции. Не буду вдаваться в детали синтаксиса, а отошлю читателя к разделу «Typedef как педагогический инструмент» в главе 6.

Функции, которым безразлично, с какими данными они работают, а интересны только указатели на данные, встречаются на удивление часто. Например, функции построения связанного списка не важно, какие данные она связывает; ей нужно лишь знать, где они находятся. Другой пример: мы можем передать указатель на функцию, то есть иметь функцию, единственная цель которой – вызвать другую функцию, передав ей аргументы, на которые известен указатель (а что хранится по этому указателю, не важно). Для таких случаев С предоставляет механизм обхода системы проверки типов – указатель на void. В объявлении

```
void *x;
```

x может указывать на функцию, на структуру, на целое и вообще на все, что угодно. В разделе «Указатель на void и структура, на которую он указывает» в главе 10 приведены примеры использования указателей на void для разных целей.

Глоссарий

Автоматическое выделение памяти – память для размещения автоматической переменной выделяется системой в точке объявления переменной и освобождается при выходе из текущей области видимости.

Автономный тест – код для тестирования небольшой части функциональности программы. См. также *Интеграционный тест*.

Библиотека – по существу, программа без функции main, то есть собрание функций, псевдонимов типов и переменных, доступных другим программам.

Булевы значения – true и false. Названы по имени Джорджа Буля, английского математика, жившего в первой половине XIX века.

Внешний указатель – см. *Непрозрачный указатель*.

Выравнивание – требование начинать элемент данных на определенной границе в памяти. Например, в случае требования о выравнивания на границу 8 бит в структуре, содержащей однобитовый элемент char[1], за которым следует 8-битовый элемент типа int, после char могут находиться 7 бит дополнения, чтобы int начинался на границе 8 бит.

Гипотеза Сепира-Уорфа – предположение о том, что язык, на котором мы разговариваем, определяет наши способности к мышлению. В самой слабой форме – очевидно, что мы часто мыслим словами. В самой сильной форме утверждает, будто мы неспособны на мысль, для которой в языке нет слов или конструкций; это заведомо не так.

Глиф – символ, применяемый для письменной коммуникации.

Глобальная переменная – переменная называется глобальной, если ее область видимости совпадает со всей программой. В действительности в C нет глобальной области видимости, но если переменная объявлена в заголовке, который с высокой вероятностью будет включаться в каждый файл с исходным кодом программы, то есть все основания считать ее глобальной.

Глубокая копия – копия структуры, содержащей указатели, куда помещаются также и данные, на которые направлены указатели.

Граф вызовов – диаграмма, состоящая из прямоугольников, соединенных стрелками, которая показывает, как функции вызывают друг друга.

Динамическое выделение памяти – выделение памяти из кучи по запросу программиста с помощью функции malloc или calloc с последующим освобождением, также по запросу, функцией free.

Закон Бенфорда – начальные цифры во многих наборах данных распределены логарифмически: *1* встречается с частотой примерно 30%, *2* – с частотой 17.5%, ..., *9* – с частотой 4.5%.

Интеграционный тест – тест, в ходе которого выполняется последовательность действий, затрагивающих несколько частей программы (для каждой из которых должен быть также написан собственный автономный тест).

---

[1] Не понятно, что имел в виду автор, говоря об «однобитовом элементе char», коль скоро тип char занимает не менее 8 бит, но пусть это останется на его совести. – *Прим. перев.*

Кадр – область в стеке, где хранится информация о функции (например, аргументы и автоматические переменные).

Квалификатор типа – описывает, как компилятор может обрабатывать переменную. Не связан с самим типом переменной (int, float и т. д.). В C допускаются только следующие квалификаторы: const, restrict, volatile, _Atomic.

Кодирование – средства, с помощью которых символы естественного языка преобразуются в числовые коды для машинной обработки. См. также *ASCII*, *многобайтная кодировка*, *широкая кодировка*.

Компилятор – строго говоря, программа, которая преобразует понятный человеку текст программы в машинные команды. Часто это слово употребляется для обозначения совокупности препроцессора, компилятора и компоновщика.

Компоновщик – программа, которая собирает вместе разрозненные части (отдельные объектные файлы и библиотеки) и согласует ссылки на внешние функции и переменные.

Куча – область памяти, из которой производится динамическое выделение. Сравните со *стеком*.

Лексема – цепочка символов, рассматриваемая как семантическая единица, например имя переменной, ключевое слово или оператор вида * или +. На первом шаге грамматического анализа текст разбивается на лексемы; для этой цели предназначены функции strtok_r и strtok_n.

Макрос – обычно короткий текст, вместо которого подставляется более длинный.

Многобайтная кодировка – кодировка текста, в которой для представления одного символа естественного языка может использоваться переменное количество байтов. Сравните с *широкой кодировкой*.

Мьютекс – структура, с помощью которой можно гарантировать, что данный ресурс в каждый момент времени использует не более одного потока.

Непрозрачный указатель – указатель на данные, структура которых неизвестна функции, получающей указатель, что не мешает ей передать указатель другим функциям, знающим, как работать с данными. Функция, написанная на скриптовом языке, может сначала вызвать C-функцию, возвращающую указатель на данные, хранящиеся в написанной на C части, а затем другая функция на скриптовом языке может воспользоваться этим указателем для работы с теми же данными.

Область видимости – часть программы, в которой переменная объявлена и доступна. Принципы хорошего стиля программирования требуют, чтобы область видимости любой переменной была как можно уже.

Оболочка – программа, позволяющая пользователю взаимодействовать с операционной системой с помощью командной строки или скриптов.

Объединение – блок памяти, который можно интерпретировать как место хранения данных нескольких типов.

Объект – структура данных и оперирующие ей функции. В идеале объект инкапсулирует некую концепцию и предоставляет ограниченный набор средств для взаимодействия с собой из внешней программы.

Объектный файл – файл, содержащие машиночитаемые команды. Обычно является результатом обработки исходного файла кода компилятором.

Отладчик – программа для интерактивного выполнения откомпилированной программы, которая позволяет пользователю приостанавливать программу, просматривать и изменять значения переменных и т. д. Часто полезна для понимания природы ошибок.

Переменная окружения – переменная, присутствующая в окружении программы. Устанавливается родительской программой, обычно оболочкой.

Плавающая точка – представление числа, близкое к научной нотации вида *2.3×10^4*, в которой имеются показатель степени (в данном случае 4) и мантисса (здесь 2.3). Если мантисса известна, то показатель степени позволяет десятичной точки «переплыть» в нужную позицию.

Подмена типа (type punning) – приведение переменной одного типа к другому типу с целью заставить компилятор рассматривать ее как объект второго типа. Например, если имеется объявление `struct {int a; char *b:} astruct`, то `(int) astruct` будет целым числом, первым элементом структуры (безопасная альтернатива обсуждается в разделе «С без зазоров» в главе 11). Часто не переносимо, всегда признак плохого стиля.

Пользовательский интерфейс – в контексте библиотеки С включает псевдонимы типов, макросы и объявления функций, призванные облегчить работу пользователя с библиотекой.

Поток – последовательность команд, исполняемая компьютером независимо от всех прочих потоков.

Препроцессор – концептуально – программа, запускаемая до компилятора для обработки директив типа `#include` и `#define`. На практике обычно является частью компилятора.

Процесс – работающая программа.

Профилировщик – программа, которая сообщает, на что ваша программа тратит время. Это позволяет сосредоточить усилия на оптимизации узких мест.

Скрипт – программа на интерпретируемом языке, например на языке оболочки.

Статическое выделение памяти – метод выделения памяти для переменных с областью видимости файла и переменных, объявленных внутри функции как `static`. Память для них выделяется до начала работы программы, и переменные существуют в течение всего времени ее работы.

Стек – область в памяти, связанная с выполнением функций. В частности, здесь хранятся автоматические переменные. У каждой выполняемой функции есть кадр в стеке. Когда вызывается новая функция, ее кадр помещается в стек над кадром вызывающей функции.

Тестовая оснастка – система для прогона последовательности автономных и интеграционных тестов. Предоставляет простые средства инициализации и освобождения вспомогательных структур, а также позволяет обнаруживать отказы, которые могут привести к (ожидаемому) останову программы.

Функция обратного вызова – функция (A), которая передается в виде аргумента другой функции (B), с тем чтобы функция B могла вызвать функцию A в ходе своей работы. Например, обобщенные функции сортировки обычно принимают функцию сравнения двух элементов.

Функция с переменным числом аргументов – функция, принимающая переменное число аргументов (например, `printf`).

Цетология – наука о китах.

Широкая кодировка – кодировка текста, в которой для представления одного символа естественного языка используется фиксированное количество байтов. Например, в кодировке UTF-32 каждый символ Unicode представлен 4 байтами. Сравните с *многобайтной кодировкой*.

ASCII – американский стандартный код для обмена информацией. Стандартное сопоставление наивным английским символам чисел из диапазона 0—127. Совет: во многих системах команда `man ascii` распечатывает таблицу кодов.

Autotools – набор программ, поставляемый фондом GNU, призванный автоматизировать компиляцию в любой системе. Включает Autoconf, Automake и Libtool.

BSD – Berkeley Software Distribution. Реализация стандарта POSIX.

Gdb – отладчик GNU.

GNU – Gnu's Not Unix.

GSL – GNU Scientific Library.

IDE – Integrated Development Environment (интегрированная среда разработки). Обычно программа с графическим интерфейсом, в основе которой лежит текстовый редактор и добавлены средства для компиляции, отладки и другие полезные программисту возможности.

Linux – строго говоря, ядро операционной системы, но чаще употребляется для обозначения всего комплекта программных средств, разработанных BSD, GNU, Internet Systems Consortium, Linux, Xorg и т. д. и собранных в унифицированный пакет.

NaN – Not-a-Number (нечисло). В стандарте IEEE 754 (представление чисел с плавающей точкой) так называется результат математически невозможной операции, например 0/0 или log(−1). Часто применяется в качестве признака отсутствующих или некорректных данных.

POSIX – Portable Operating System Interface (переносимый интерфейс операционных систем). Разработанный IEEE стандарт, которому должны удовлетворять UNIX-подобные операционные системы. Описывает состав функций в библиотеке C, оболочку и некоторые важнейшие утилиты.

Pthread – поток POSIX. Поток, созданный с помощью интерфейса к потокам из C, определенного в стандарте POSIX.

RNG – генератор случайных чисел. Слово «случайный» здесь означает, что между членами генерируемой последовательности чисел нет систематической зависимости.

RTFM – читай руководство.

Segfault – нарушение защиты памяти. Обращение к недоступному или несуществующему адресу в памяти. Приводит к немедленному завершению программы операционной системой и потому – в силу общности последствий – часто употребляется для обозначения любой ошибки, влекущей за собой аварийный останов программы.

SHA – Secure Hash Algorithm. Алгоритм вычисления криптографической контрольной суммы (хэша).

SQL – Structured Query Language (структурированный язык запросов). Стандартизованное средство взаимодействия с базами данных.

UTF – Unicode Transformation Format.

XML – Extensible Markup Language (расширяемый язык разметки).

Библиография

[abelson-1996] Abelson, H., G. J. Sussman, and J. Sussman (1996). *Structure and Interpretation of Computer Programs.* The MIT Press.

[Breshears 2009] Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications.* O'Reilly Media.

[Calcote 2010] Calcote, J. (2010). *Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool.* No Starch Press.

[Deitel 2013] Deitel, P. and H. Deitel (2013). *C for Programmers with an Introduction to C11 (Deitel Developer Series).* Prentice Hall[1].

[Dijkstra 1968] Dijkstra, E. (1968, March). Go to statement considered harmful. *Communications of the ACM 11* (3), 147—148.

[Friedl 2002] Friedl, J. E. F. (2002). *Mastering Regular Expressions.* O'Reilly Media[2].

[Goldberg 1991] Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23* (1), 5—48.

[Goodliffe 2006] Goodliffe, P. (2006). *Code Craft: The Practice of Writing Excellent Code.* No Starch Press.

[Gough 2003] Gough, B. (Ed.) (2003). *GNU Scientific Library Reference Manual* (2nd ed.). Network Theory, Ltd.

[Gove 2010] Gove, D. (2010). *Multicore Application Programming: for Windows, Linux, and Oracle Solaris (Developer's Library).* Addison-Wesley Professional.

[Grama 2003] Grama, A., G. Karypis, V. Kumar, and A. Gupta (2003). *Introduction to Parallel Computing (2nd Edition).* Addison-Wesley.

[Griffiths 2012] Griffiths, D. and D. Griffiths (2012). *Head First C.* O'Reilly Media.

[Hanson 1996] Hanson, D. R. (1996). *C Interfaces and Implementations: Techniques for Creating Reusable Software.* Addison-Wesley Professional.

[Harbison 1991] Harbison, S. P. and G. L. Steele Jr. (1991). *C: A Reference Manual* (3rd ed.). Prentice Hall.

[Kernighan 1978] Kernighan, B. W. and D. M. Ritchie (1978). *The C Programming Language* (1st ed.). Prentice Hall[3].

[Kernighan 1988] Kernighan, B. W. and D. M. Ritchie (1988). *The C Programming Language* (2nd ed.). Prentice Hall[4].

[Klemens 2008] Klemens, B. (2008). *Modeling with Data: Tools and Techniques for Statistical Computing.* Princeton University Press.

[Kochan 2004] Kochan, S. G. (2004). *Programming in C* (3rd ed.). Sams.

---

[1] Дейтел П., Дейтел Х. С для программистов с введением в С11. – М.: ДМК Пресс, 2014.
[2] Фридл Д. Регулярные выражения. – СПб.: Символ-Плюс, 2008.
[3] Керниган Б., Ричи Д. Язык программирования Си. – М.: Финансы и статистика, 1992.
[4] Керниган Б., Ритчи Д. Язык программирования Си. – 2-е изд. – М.: Вильямс, 2015.

[van der Linden 1994] van der Linden, P. (1994). *Expert C Programming: Deep C Secrets.* Prentice Hall.

[Meyers 2000] Meyers, S. (2000, February). *How non-member functions improve encapsulation.* C/C++ Users Journal.

[Meyers 2005] Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley Professional[1].

[Nabokov 1962] Nabokov, V. (1962). *Pale Fire.* G P Putnams's Sons.

[Norman 2002] Norman, D. A. (2002). *The Design of Everyday Things.* Basic Books[2].

[Oliveira 2006] Oliveira, S. and D. E. Stewart (2006). *Writing Scientific Software: A Guide to Good Style.* Cambridge University Press.

[Oram 1991] Oram, A. and Talbott, T (1991). *Managing Projects with Make.* O'Reilly Media.

[Oualline 1997] Oualline, S. (1997). *Practical C Programming* (3rd ed.). *O'Reilly Media.*

[Page 2008] Page, A., K. Johnston, and B. Rollison (2008). *How We Test Software at Microsoft.* Microsoft Press.

[Perry 1994] Perry, G. (1994). *Absolute Beginner's Guide to C* (2nd ed.). Sams.

[Prata 2004] Prata, S. (2004). *The Waite Group's C Primer Plus* (5th ed.). Waite Group Press[3].

[Press 1988] Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling (1988). *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press.

[Press 1992] Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling (1992). *Numerical Recipes in C: The Art of Scientific Computing* (2nd ed.). Cambridge University Press.

[Prinz 2005] Prinz, P. and T. Crawford (2005). *C in a Nutshell.* O'Reilly Media.

[Spolsky 2008] Spolsky, J. (2008). *More Joel on Software: Further Thoughts on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity.* Apress[4].

[Stallman 2002] Stallman, R. M., R. Pesch, and S. Shebs (2002). *Debugging with GDB: The GNU Source-Level Debugger.* Free Software Foundation.

[Stroustrup 1986] Stroustrup, B. (1986). *The C++ Programming Language.* Addison-Wesley[5].

[Ullman 2004] Ullman, L. and M. Liyanage (2004). *C Programming.* Peachpit Press.

---

[1] Мейерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2014.

[2] Норман Д. Дизайн привычных вещей. – М.: Манн, Иванов и Фарбер, 2013.

[3] Прата С. Язык программирования С: лекции и упражнения. – М.: Вильямс, 2013.

[4] Спольски Джоэл Х. И снова о программировании. – СПб.: Символ-Плюс, 2009.

[5] Страуструп Б. Язык программирования C++. – М.: Бином, 2011.

Предметный указатель

Символы

A

B

K

L

M

U

V

W

X

Z

А

Б

В

П

Р

С

Т

У

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью), по которому должны быть
высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Бен Клеменс

Язык C в XXI веке

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 23,5. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

«Для разработки на С вы используете только vi и cc? Управление памятью вызывает такие же страдания, как в 90-х годах прошлого века? Бен Клеменс со знанием дела рассматривает эти и другие типичные проблемы, демонстрируя, как инструменты, призванные облегчить жизнь программиста на С, помогают отлаживать код, находить утечки памяти, организовывать процесс компиляции и управлять версиями программы.»

— Дэйв Китабян,

Руководитель программных разработок, NetCarrier Telecom

Выбросьте на свалку старые представления о С и познакомьтесь с языком, который давно уже вырос из детских штанишек. Здесь вы узнаете о современных приемах программирования, о которых не рассказывают в других учебниках по С. И новичок, и ветеран, решивший освежить старые знания, откроет для себя что-то новое.

Язык С — не просто фундамент всех современных языков программирования, он и сам — современный язык, идеальный для написания эффективных приложений передового уровня. Забудьте об идиомах, когда-то придуманных для старых мэйнфреймов, и изучайте инструменты, которые пригодятся для работы с этим преобразившимся и вызывающе простым языком. И неважно, какой язык программирования у вас в фаворе сейчас — очень скоро вы поймете, что в XXI веке С по-прежнему играет рок.

- Обустройство среды программирования на С с помощью средств оболочки, файлов makefile, текстовых редакторов, отладчиков и инструментов проверки памяти;
- использование Autotools, кросс-платформенного сборщика пакетов, ставшего стандартом де факто для С;
- спорные концепции С, слишком полезные, чтобы расстаться с ними;
- решение проблем построения строк с помощью функций, описанных в стандарте С;
- использование современных синтаксических конструкций для функций, принимающих структуры;
- создание высокоуровневых объектных библиотек и программ;
- выполнение математических расчетов, взаимодействие с Интернет- серверами и обращение к базам данных с помощью существующих библиотек, написанных на С.

В это издание включен новый материал о параллельных потоках, виртуальных таблицах, числовых типов из стандарта С99 и других средствах.

Бен Клеменс (Ben Klemens) *занимался статистическим анализом и вычислительно-сложным моделированием в интересах Брукингского института, Всемирного банка, Национального института психического здоровья и правительства США. Он также работал в Брукингском институте и в Фонде свободного программного обеспечения и многое сделал для того, чтобы авторы сохраняли права на написанное ими программное обеспечение. В настоящее время возглавляет группу статистических расчетов в исследовательском подразделении Бюро переписи населения США.*

Интернет-магазин:
www.dmkpress.com



O'REILLY®

www.дмк.рф

ISBN 978-5-97060-101-3

